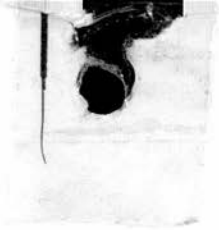

The Use of Proof Plans in Tactic Synthesis

Jason Gallagher

Ph.D.

University of Edinburgh

1993

Abstract

We undertake a programme of tactic synthesis. We first formalize the notion of a tactic as a rewrite rule, then give a correctness criterion for this by means of a reflection mechanism in the constructive type theory OYSTER. We further formalize the notion of a tactic specification, given as a synthesis goal and a decidability goal. We use a proof planner, CL^AM, to guide the search for inductive proofs of these, and are able to successfully synthesize several tactics in this fashion. This involves two extensions to existing methods: context-sensitive rewriting and higher-order wave rules. Further, we show that from a proof of the decidability goal one may compile to a Prolog program a pseudo-tactic which may be run to efficiently simulate the input/output behaviour of the synthetic tactic.

I declare that this thesis has been composed by myself and that the work described in it is my own:

Jason Gallagher

Acknowledgements

I would like to thank my supervisors Professor Alan Bundy and Dr Andrew Ireland for their advice, support and patience throughout this research project. I would also like to express my gratitude to past and present members of the Mathematical Reasoning Group at Edinburgh for providing such a good working and research environment, and in particular to Andrew Stevens, Ian Green, Ian Gent, Helen Lowe and Francisco Cantu Ortiz for helpful feedback on my thesis drafts. In addition I gratefully acknowledge the support of a Science and Engineering Research Council studentship.

This thesis is dedicated to my parents.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Aims and Motivation

A tactic is a program which performs a theorem proving task, usually of a specific, intended kind, for example applying an inference rule or rewriting a subterm (Gordon *et al*, 1979). This concept is explained in detail in section 2.2. Our aim in this thesis is to set up a framework and develop techniques whereby we are able to use a theorem prover and formal proof system to synthesize tactics, and to partially automate this. Our original motivation for this arose from the proof planner we will later use: *CLAM*, developed by the Maths Reasoning Group (MRG) at Edinburgh University (vanHarmelen *et al*, 1993). The philosophy behind *CLAM*, which is explained in more detail in appendix A, is that the search problem in theorem proving is best tackled by reasoning in a similar way to humans: develop tactics which are specific to a given class of problem and use meta-level information about the goal and about the tactics themselves to apply these appropriately. *CLAM* does this by using *methods*, where one may write:

$$\text{method} = \text{tactic} + \text{specification}$$

Thus, our original aim was to automate the synthesis of the tactic part of a method given its specification. At present the tactic parts of *CLAM*'s methods are written by hand, to carry out in the object logic what is promised in specification.

However, tactic verification and synthesis is not only useful with respect to *CLAM* methods; some consider it an essential step if we are ever to use theorem provers for real world problems. (Knoblock & Constable, 1986) have the following to say about tactics in the NuPRL (Constable *et al*, 1986) proof environment:

> Tactics raise the level of reasoning in the system, and for that they are vital. But if tactics must be executed, producing explicit primitive-level proofs, it becomes impossibly costly to raise the level of reasoning very far.... And yet we may know that if run they will succeed. In this case one approach is to determine *analytic conditions* that guarantee success of a tactic and then *prove formally in the system* that it works when the conditions are met. If the conditions can be easily checked or proved, then great computational savings are possible. Such an approach will allow the system to achieve high levels of abstraction. [My italics.]

The last point can be illustrated by considering what kind of knowledge we can represent in a proof system. We usually think of stating formally such things as definitions and theorems and their proofs, but the theorem prover or tactics we used to produce the latter, while formal parts of the system, are not part of the object-level. Thus, while we can state and use knowledge which can be expressed in the form of definitions and theorems, we can only give a sketchy, informal idea of knowledge which must be expressed in the writing of a tactic or theorem prover. By formalising a partial meta-theory as proposed by Constable and Knoblock we greatly enhance our ability to represent and use this "deeper knowledge". A good example of these different types of knowledge is the following: at the object-level we may prove theorems such as the associativity, commutativity and idempotence of the connective $\leftrightarrow$. In the meta-theory we can state the fact, and with a suitable reflection mechanism prove, that: if each distinct propositional variable occurs an even number of times in an expression built only with $\leftrightarrow$ (the analytic precondition), then it is a theorem. This example can be found in (Weyhrauch, 1980)[p. 146].

The reason for using a theorem prover in an attempt to partially automate tactic synthesis is succinctly put by Boyer and Moore in (Boyer & Strother Moore, 1981):

> If it is not practical to prove the correctness of new procedures with the tools provided then... the extensibility is either unusable or unsafe

because the users will add axioms stating the correctness of the new procedures.

We turn tactic synthesis into a theorem proving task by use of constructive type theory. In this setting, by proving a goal of the form

$$\vdash \forall x : input.\exists y : output.spec(x, y) \tag{1.1}$$

we may extract a program which satisfies the specification *spec*. *CLAM* method specifications are broken up, as in (Knoblock & Constable, 1986), into a precondition and effect. Thus our synthesis theorems typically have the form:

$$\vdash \forall x : input.\exists y : output.precond(x) \rightarrow effect(x, y)$$

and it is proof of these theorems which we are principally interested in automating. We also attempt to prove theorems asserting the decidability of the pre-conditions, i.e.:

$$\vdash \forall x : input.precond(x) \vee \neg precond(x)$$

With these proofs one may transform using the tactics embodied in the synthesis theorems into simply evaluating the pre-conditions (see chapters 4 and 5).

Of course, program synthesis, and tactic synthesis in particular, is a very hard problem. To talk meaningfully about this we must first say what programming language we are using, what language a specification is written in, and what it means for a program to meet a specification. These topics are discussed in detail in chapter 2, but under some very humble assumptions the problem of program synthesis is seen to be *non-computable*.

The correctness of a program is sometimes split into two parts: *partial correctness* given that it terminates, the program meets its specification and *termination* that it eventually halts returning an answer. Assume that our programming language is rich enough to express any general recursive function and that our specification language is sufficient to assert that for a given input an (encoding of a) program terminates and that two (encodings of) programs have the same

input/output behaviour[1]. Turing showed with the halting problem (Turing, 1936) that even the termination part of program verification is in general undecidable. But the problem of program synthesis subsumes that of verification in the following sense. For all (encodings of) specifications $spec(input, output)$, an algorithm for program synthesis would decide whether there existed a program, (with encoding) $P(x)$, meeting *spec* and if so produce P. To *verify* that a given P met a given specification *spec* we would apply the program synthesis algorithm to the specification

$$spec'(i, o) \equiv spec(i, o) \wedge P(i) = o$$

If the algorithm synthesized a program P' we would know that P met *spec*; if it indicated that no such P' existed we would know that P did not meet *spec* since otherwise P itself would contradict the non-existence.

In the more specific setting of OYSTER type theory we say that an algorithm solves the program synthesis problem if it can decide (and prove in the affirmative case) all conjectures of form (1.1). An algorithm is said to solve the program verification problem if it can decide (and prove in the affirmative case), for all well-typed functions $f : input \rightarrow output$, all conjectures of the form:

$$\vdash \forall x : output.spec_v(x, f(x))$$

An algorithm for the program verification problem can be reduced to one for the synthesis problem by putting

$$spec(x, y) \equiv y = f(x) \ \texttt{in} \ output \wedge spec_v(x, y)$$

If there is no general solution to the problem of tactic synthesis, how ambitious should our programme be? The state-of-the-art in inductive theorem proving is generally taken to be the Boyer and Moore theorem prover, NQTHM (Boyer & Moore, 1979). In (Boyer & Strother Moore, 1981), they were able to

---

[1] OYSTER (Horn, 1988) and NuPRL (Constable *et al.* 1986) have both these properties.

verify a procedure for cancelling addition across an equation after stating and proving three intermediate lemmas (see section 6.2). We should therefore have achieved something significant if we reach a similar level of user-assistance in synthesis proofs. Boyer and Moore summarize:

> But we do not see extensibility as a panacea for the current lack of theorem-proving power. It is a solution to a relatively simple problem: how to obtain insurance against unsoundness. The truly hard intellectual problem remains: the discovery of harmoniously cooperating heuristics for marshalling a very large number of facts and constructing difficult proofs.

We now outline our programme which constitutes the major part this thesis. Before we can begin to synthesize tactics we have the following prerequisites:

a formalization of the notion of tactic Before we can do anything else we must say in a rigorous way exactly what is meant by "tactic".

a formal means of specifying tactics We need a language sufficiently expressive that one can represent and reason about such objects as terms, proofs and tactics, and into which we can naturally translate our specifications.

formalized a notion of correctness As well as meeting a specification, we need a means of ensuring that our synthetic tactics behaved soundly with respect to our object logic.

a theorem prover and techniques for carrying out the synthesis proofs As Boyer and Moore note above, without some tools to help in the synthesis, a formalization in any system is unusable. Using the $CLAM$ proof planner for this purpose is the major contribution of this thesis.

a means of using the synthesized tactic efficiently Automated theorem proving is a particularly pragmatic discipline; our work is of little benefit if it is of no practical use.

We have chosen to formalize tactics in a fashion well known in the literature (Boyer & Strother Moore, 1981; Howe, 1988a): a partial reflection mechanism.

This allows us to give a strict notion of correctness in terms of preserving the meaning of a term. In order to bring the resulting synthesis proofs within the scope of *CLAM* we have made a number of simplifications as to what constitutes a tactic in our formalism. The first simplification arises from our use of the constructive type theory OYSTER (Horn, 1988) as the object logic. The tactics are derived from the extracts of the synthesis proofs and are therefore *total*; they may fail, in the same way some programming languages allow exceptions to be raised, but will always terminate. The second simplification arises from the generality of tactics — tactics can be written to perform any algorithmically definable derivation and would therefore require a full description of the object logic, for example a complete reflection mechanism, to supply a notion of correctness. A much simpler *partial* reflection mechanism gives an immediate notion of correctness if we require that the input and output of a tactic have the same meaning, but for this definition to be applicable we require that a tactic have exactly one output, i.e. that it be a rewrite tactic. Our third simplification is in restricting the class of terms we attempt to rewrite to contain function symbols whose arity is at most 2 and that these are over the domain of natural numbers. We show that this imposes no theoretical loss of generality and is still useful in practice.

So, in essence we equate tactics with functions which operate on the meta-level provided by our partial reflection mechanism. How are these of use in practice and why use them at all? To motivate this we again quote Knoblock and Constable:

> This has two key advantages. We can prove in NuPRL itself that a meta-function will produce the desired proof; so it may not be necessary to explicitly produce the proof (unless it is needed for computation). This leads to a more efficient application of the tactic method[2]. Second, it eliminates the need to express meta-theoretic concepts in a distinct language. Thus there is only one language to learn, the NuPRL theory.

Thus the motivation is to compile object-level proof obligations into a once-only correctness proof (as part of the synthesis), leaving only the "analytic conditions"

[2]in the same way that cutting in an already-proven lemma does not require its reproving.

to be shown each time the tactic is used. This exemplifies half the approach of *CLAM*: the specification part of a method. Of course, *CLAM* has to run a "real" tactic to meet its object-level proof obligations. Synthesized tactics, on the other hand, can be used without further proof. Because of this we concentrate on tactics where the ratio of object-level cost/meta-level cost is large. In the case where this is not so we show how to make formal tactics informal by compiling them to a language such as Prolog or ML.

Since tactics are, by their nature, recursive (in the non-logical sense) functions, any non-trivial synthesis proof will require a degree of inductive reasoning. It is this aspect of automation which interests us most, and for this reason we use as our theorem prover *CLAM* itself. We show that, by means of various extensions to the existing set of methods for general purpose inductive theorem proving, we are able to apply *CLAM* to tactic synthesis and give a coherent "rippling story" as an overall, controlling *proof plan* (see chapter 4). Our extensions/additions include: the partial reflection mechanism implemented in OYSTER(chapter 3); higher-order wave rules (section 4.8); context-sensitive rewriting (section 4.7); decidability proofs (section 4.6); compilation of synthesized tactic to Prolog and ML (chapter 5); guidance using the correctness goal (section ??); and load-time simulation of *CLAM* (section 3.5).

We shall assess in chapter 7 how far we have come to meeting the goals outlined above. As the ability of automatic provers, and *CLAM* in particular, increases, so will the sophistication of the tactics we shall be able to synthesize automatically. We give here some of the implications of automating tactic synthesis as posited:

Ordinary use We have the immediate benefit of not having to hand-write tactics to meet meta-level specifications. We also know that the synthesized tactics will always work.

Reflective use We have the additional benefits of efficiency where work is largely at the meta-level. This will become increasingly important as deeper knowledge representation becomes important in automated theorem proving.

Bootstrapping The circularity of the above programme, whereby a theorem prover proves theorems which add to its theorem proving ability, is exemplified in an existing system (Boyer & Strother Moore, 1981).

Methodology We hope that the need to rigorously formalize the notion of tactic will lead to a better understanding of how best to provide a formal metalanguage for this purpose, and to the systematic use of that language.

Abstract planners Proof planners like *CLAM* should be (almost) independent of particular proof systems. By limiting the amount of machinery that needs to be ported in order to use the proof planner, i.e. by implementing at the object level via a reflection mechanism a large proportion of the tactics we use, we come closer to that ideal.

1.2 Formal vs informal

Before concluding, we briefly explain our reasons for eschewing what may seem a much more obvious and simple solution to our goal.

A *CLAM* method consists of two parts: a meta-level description, or specification, of the behaviour of the method, and a tactic to realise this behaviour at the object-level. Our goal was, given only the first part of a method, to (semi-)automatically synthesize a suitable tactic. Let us give an example. A method called `eval_def/2` (all *CLAM* terminology is explained in the appendix A) has the job of checking whether an expression in the goal is an instance of a definition which can be unfolded. We give a simplified version here ignoring such features as quantification and meta-variables in the goal: A method, as in this example, has six slots; the specification part is given by slots 1 5 and the tactic by slot 6. The first slot gives the method's name, here `eval_def/2`, with arguments for the position within the goal of the subterm to be unfolded (`Pos`) and the name of the theorem embodying the definition (`Rule`). Such a theorem is the step case of the definition of `plus`

```
method(eval_def(Pos,Rule),
       H==>G,
       [exp_at(G,Pos,Exp),
        func_defeqn(Exp,Exp,NewExp),
       ],
       [replace(Pos,NewExp,G,NewG)
       ],
       [H==>NewG],
       eval_def(Pos,[Rule,Dir])
      ).
```

Figure 1–1: A simplified method

embodied in the theorem `plus2`:

$$\vdash \forall x, y : \texttt{pnat.plus}(\texttt{s}(x), y) = \texttt{s}(\texttt{plus}(x, y)) \texttt{ in pnat}$$

The second slot is an *input* template matching the goal. Since OYSTER is a sequent-based logic this consists of a hypothesis list (`H`) and a conclusion (`G`). Third is a list of *preconditions*. These are runnable Prolog goals all of which must succeed if the method is to be deemed *applicable* to the goal. Here, every subexpression (`Exp`) of the goal is tested to see whether it is an instance of the left-hand side of some definition. When a method is called by the planner it is usually done so with uninstantiated arguments. Execution of the preconditions typically results in their instantiation, in this case completely so; internal variables may also become instantiated. The fourth slot, the *effects*, is another list of runnable Prolog goals. These carry out the actions necessary to construct any new subgoals and determine any remaining variables; all effects should succeed if the method is applicable. The fifth slot, the *output*, is a list of subgoals, in this example a single sequent consisting of the original goal with the definition instance replaced by its unfolding.

The sixth slot is the name of a tactic which *should* implement at the object-level, i.e. in OYSTER, the behaviour described at the meta-level in slots 1 5. That is: slot 6 should name a tactic in the form of a runnable Prolog goal (which is how

tactics are implemented in OYSTER) which applied to a goal identical to the input (slot 2) results in subgoals identical to those in the output (slot 5).[3] In our simple example the tactic could consist, for instance, of cutting in the theorem called `Rule`, suitably instantiating it to give a new hypothesis of the form

$$New = NewExp$$

and using this to justify an application of OYSTER's substitution rule primitive.

From our description of how a method is used at the meta-level by a planner and its tactic part executed at the object-level, it can be seen there is no formal (in the sense of an OYSTER proof) connection between slots 1 5 and slot 6. When a *CLAM* user comes to write a new method (s)he is responsible for ensuring that the tactic of slot 6 fulfills the promise made in slots 1 5 in the same way that a programmer is responsible for ensuring that a piece of code meets a client's specifications. This is an onerous and tedious task where mistakes are common, especially in a domain as complicated as proofs. Computer scientists believe that *formal methods* offer the best hope for building correct software. A *CLAM* method might already be said to give a rigorous, rather than a formal, specification of the required tactic; can we not use this to eliminate the need to write tactics by hand?

There appear to be two distinct routes to achieving this, which we shall call the *formal* approach and the *informal* approach. We describe the latter first.

The new subgoals of slot 5 are constructed by the effects slot of the method. With each predicate of the meta-language we could associate a tactic to carry out an analogous operation at the object-level. In our example one might associate a substitution tactic with `replace_at/4`. By suitably combining the associated tactics one may be able to realise a tactic satisfying the method. For example we might have:

---

[3]Note: there is no name clash between the method **`eval_def/2`** and the tactic **`eval_def/2`** since the method is implemented as the six-part data structure described above and used by the planner; the method is not itself a Prolog predicate. The same is true of methods and tactics in general.

```
eval_def(Pos,Rule):-
        hyp_list(H),                    %Get current hypotheses
        goal(G),                        %and conclusion.
        exp_at(G,Pos,Exp),              %Run the pre-conditions.
        func_defeqn(Exp,Exp,NewExp),    %Ditto.
        replace(Pos,NewExp,G,NewG),     %Run the effects.
        apply(...subst_tac(...)...).    %Apply substitution tactic
                                        %associated with replace_at/4.
```

However, there are several problems with this approach. First, as its name suggests, it is informal. We may have a correctness proof for some algorithm on paper, but there is no way to exploit this internally in OYSTER we can never prove in OYSTER that this tactic satisfies slots 1 5. Second, it is not clear how to "suitably combine" the various tactics associated with each of the effects to give the full tactic. Effects are linked by logical variables to each other and to all the other slots, and often these variables don't correspond to well-formed object-level objects, e.g. in the example above `Exp` and `NewExp` are terms but not propositions. Third, the approach appears difficult to modularize or scale up. Each predicate of the method language would need a complicated, description which would depend on all the other descriptions and where the simple logical semantics is lost. This approach also suffers from fourth problem, in common with the formal approach, that any annotation used by the method must somehow be accommodated transparently at the object level. For example, if wave annotation (see glossary) were used by the method, we would have to represent it at the object-level in such a way as to leave the meaning of any sequent unchanged.

Alternatively there is what we have called the formal approach. First, a *reflection mechanism* (see chapter 3) is used to formalize the object-level, or part of it, inside itself. Using this, a formal, object-level definition of the predicates of the method language is given. In this setting theorems can be proved, for example, asserting the existence, given a suitable input, of a suitable subgoal in essence a tactic. One might prove the theorem:

$$\vdash \forall g, exp, nexp : metaterm, pos : posn, rule : name.$$

$$exp\_at(g, pos, exp) \wedge func\_defeqn(exp, rule, nexp) \rightarrow$$
$$\exists ng : metaterm.replace\_at(pos, nexp, g, ng) \wedge g \sim ng$$

The expression $g \sim ng$ is part of the reflection mechanism and ensures that the original goal and the subgoal "mean" the same thing, that the original goal follows from the new one. Such a theorem can be used as a tactic when the proof is carried out in a constructive logic such as OYSTER. By instantiating with suitable terms and *proving* the pre-conditions, one can extract entirely automatically a term ng and a proof that $g \sim ng$. The latter can be used to replace the original goal g with the subgoal ng.

This approach, too, has problems, apart from the common problem mentioned above of representing any annotation used by the methods. Whereas in the informal approach we tried to treat formal objects (tactics) informally, here we attempt to treat informal objects (the method language) formally. Setting up a reflection mechanism and formal definitions of each of the method language predicates is, like writing methods, a delicate and time-consuming task; unlike method writing, it only needs to be done once. There is also a better prospect for the formal approach: it is possible, and indeed quite practicable, to move a large part of the informal aspects of a proof environment over to the object-level (see chapter 6 for examples). There is little point, however, in a proof system with informal tactics. Should we wish to recover a stand-alone tactic from a proof, one which can be used in the absence of the reflection mechanism and thus is no longer formally correct, we show in chapter 5 that this is still possible.

We have chosen in this thesis to take the formal approach. This necessitates that we first choose an appropriate means of specification and representation. These choices and design decisions are discussed in chapters 2 and 3. The use of a reflection mechanism and the representation of the method language at the object-level give rise to long and intricate proof obligations. Fortunately, much of the work is of a kind ideally suited to automation via a mechanical theorem prover as we see in chapter 4.

1.3 Structure of Thesis

The thesis is comprised of seven chapters of which we now give a short overview. Chapter 1 introduced the ideas propounded in the rest of the thesis, explaining the goals of and motivation for the work.

Chapter 2 examines the program specification methodologies representative of the current state-of-the-art. These are analysed with regard to their usefulness and amenability to our programme. Such a survey is required to justify our eventual choice of constructive type theory. We describe at the end of chapter 2 the proof system (OYSTER) and proof planner ($CLAM$) used in the sequel.

Chapter 3 makes concrete our use of type theory as constituted in the OYSTER system. In it we describe how we go about representing in type theory the data and functional objects commonly used when programming tactics. The motivation for such a representation is a partial reflection mechanism giving rise to a precise notion of tactic correctness.

In chapter 4 we explore the use of the $CLAM$ proof planner in tactic synthesis. We divide the process into synthesis of atomic tactics from their specifications, and the putting together of existing tactics to form super tactics. In order to do this we have introduced several new techniques into $CLAM$ in particular the use of context sensitive rewriting and higher-order wave rules.

Besides the use of the reflection mechanism of chapter 3 in "running" the synthesized tactics we also examine, in chapter 5, the "compilation" of synthesized tactics into traditional programming languages (ML and Prolog). With our use of the notion of decidability to simulate logic programs this is of interest in its own right, but may also lead to substantial speed-ups in some cases.

The thesis is concluded by a chapter describing related work and comparing it to ours (chapter 6), and conclusions in chapter 7: how far we have gone to meeting our original goals, what the outlook is for our programme and what interesting research problems remain.

Source code and OYSTER and *CLAM* objects referred to in the text are gathered into appendices. We have implemented to a greater or less extent all programming mentioned in this thesis. However, in some cases the implementation is only partial, carried out sufficiently far to show feasibility. While the reflection mechanism of chapter 3 and the tactic compilation algorithm in its limited form (chapter 5) are completely implemented for example, only some of the methods and submethods required for a full *CLAM* implementation employing the ideas of chapter 4 have been written: some of the example plans and proofs in chapter 4 have been simulated by hand.

Chapter 2

Specification

2.1 Introduction

As has already been pointed out, tactics are a special kind of program, ones which manipulate proofs. We define what is meant formally by the term *tactic* in section 2.2. Later in this section we give an looser, intuitive idea of what a tactic is, how they are used, and what minimal properties we believe they should possess. Our main goal is to verify and synthesize these special programs, and in order to do this we first need a way of describing a program's behaviour: a specification. In verifying a given program we must show that it satisfies a given specification; program synthesis requires that we satisfy a given specification by explicitly producing a program which does so. In this chapter we survey and summarize some of the existing methods of program specification and their concomitant methods of program development, examining them in relation to a number of criteria important to our goal, and justifying the method we finally choose.

There are two properties which we consider it essential that a tactic possess. These relate to its use as a pseudo-inference rule in proofs, where it maps a conclusion to premises (a backward rule). From the dictum that "we recognize a proof when we see one" (Sundholm, 1983, p. 156), it follows that all proof rules should be decidable and that the premises of a rule should be uniquely determined by its

conclusion and possible side arguments. Thus, a proof rule, and therefore a tactic, should be *total* and *deterministic*. In practice tactics in existing systems have no such restrictions and may involve search where these properties do not hold. For example, a tactic which implements a complete resolution theorem prover for first-order predicate logic can never be more than semi-computable. However, we will assume that *our* tactics are incomplete but "useful" in a particular domain and that, if unsuccessful, they terminate with an indication of failure rather that not at all. Assuming such properties greatly simplifies the machinery required to specify and reason about tactics. Note that circumscribing tactics in this way does not restrict them to being merely rule macros: tactics may use meta-level information and their behaviour may depend in a complicated way on the form of the goal. A good example of this is the tactic mentioned in chapter 1 which simplified $\leftrightarrow$-expressions. Having said that tactics can terminate with failure we shall therefore need to be able to specify this kind of behaviour also.

The above analysis examined the properties that an individual tactic should possess. We also need to think about how tactics will be used together and allow for future changes and improvements to different parts of the system. These are the so-called programming-in-the-large aspects of this work. There are two approaches to this, sometimes termed "black box" and "glass box" development. In the former, as the separate modules of a program are developed the only assumptions that one module may make of another are those given by the specification. In the glass box approach, as well as the information provided by the specification, one may also use the implementational details, for example that an abstract data structure has a certain concrete representation. This latter approach, while possibly resulting in terser code, carries with it the danger that as different parts of the system are changed, the whole may no longer work. This re-usability aspect is important in our work since, as *CLAM* updates arrive, so we may choose to re-synthesize a whole suite of tactics. If the latter is to continue to work we must stipulate a *black box* development process.

2.2 Tactics, tacticals and conversions

As was noted in chapter 1, one of our prerequisites is a formalization of the notion of tactic. Before examining the means available for doing this it is necessary that we first say exactly what is meant by the term "tactic" and how they are implemented in existing systems. For a more detailed explanation the reader is referred to the literature cited.

Tactics arose out of the need, encountered during the course of developing a proof system for the PPλ logic, for a "framework in which a user can both design his own partial proof strategies and execute single steps of proof" (Gordon *et al*, 1977a). The result of this work was the proof system LCF with its own metalanguage, ML, for programming tactics and more generally manipulating the object logic. In discussing the requirements of a tactic language (Gordon *et al*, 1977a) say:

> The principal aims in designing ML were to make it impossible to prove non-theorems yet easy to program strategies for performing proofs. A strategy or recipe for proof could be something like "induction on f and g, followed by assuming antecedents and doing case analysis, all interleaved with simplification." ... The point is that such strategies appear to be built from simpler ones (which we call tactics) by a number of general operations in fairly regular ways; we call these operations tacticals ...
>
> For programming tactics and tacticals ... the following ingredients in ML were soon found to be expedient (almost necessary): the ability to handle higher-order functions, a rigorous but flexible type structure, a mechanism for generating and trapping failures, and an abstract representation of the object language.

As will be explained later in chapter 3, we have met these needs with respectively the λ-calculus of OYSTER, its type theory, the use of the union type `?t`, and a reflection mechanism. We first describe, in some detail, how tactics are implemented in the NuPRL system (Constable *et al*, 1986), most of which is a specialization of the LCF concept (Gordon *et al*, 1977b).

The original implementors of LCF took an abstract view of the process of proving theorems by defining a `goal` type, a member of which was *achieved* by a suitable

member of the `shot` type. A *tactic* was a program used to accomplish backwards proof by breaking a goal into a list of subgoals and providing a `validation` mapping shots achieving the subgoals to a shot achieving the goal. Thus we have the declarations:

```
lettype validation = shot list -> shot;;
lettype tactic = goal -> goal list # validation;;
```

The types `goal` and `shot` are ready-defined ML abstract types. Implementing an object logic entails defining a notion of `goal`, and a notion of inference encoded in `shot` this is the interface at which the object logic and ML meet. By allowing the user access to all the representation (destructor) functions of the `shot` type, so that he may analyse theorems syntactically, but allowing access to the abstracting (constructor) functions *only* via encodings of the primitive inference rules, soundness is ensured. If an illegal instance of the latter is invoked the result is an ML *failure* trap; this can either be caught and acted on or result in a top-level failure.

NuPRL is an instance of this approach. Both `goal` and `shot` here become the type `proof` of partial proofs. Members of `proof` consist of a sequent, a refinement and a list of subproofs of the children of the refinement. The latter two may be missing in some leaf nodes of an incomplete proof. A member of `proof` achieves itself, with a partial proof considered complete when all its leaf nodes are childless. As described above, soundness is ensured by having only one[1] primitive function in ML that constructs new proof objects: `refine`, of type `rule -> tactic`.

Tactics are used in practice like derived inference rules, in the following way[2]. According to (Constable *et al*, 1986): at a particular point in a proof the ML variable `prlgoal` is associated with the current node, a member of `proof`; the named

---

[1] This substantiates the criticism of (Knoblock & Constable, 1986) quoted in chapter 1 that all proofs must ultimately be carried out at the primitive level.

[2] There are actually two kinds of tactic used in NuPRL: *refinement tactics* described here, and *transformation tactics* which we shall not discuss.

tactic is applied to `prlgoal`, producing a (possibly) empty list of subgoals and a validation; the validation is applied to the subgoals; the tactic name is installed as the name of the refinement rule for the current node; and the subgoals become the children of the current node. From this description it should be clear that tactics are used in a *functional, side-effect-free* manner, though they may make extensive use of global data such as other tactics and already proven theorems.

A user is able to define his own tactics by writing the appropriate ML programs. An example is the identity tactic, `IDTAC`:

```
let IDTAC (g:proof) = [g],hd;;
```

This returns the original goal, `g`, as the sole subgoal and uses `hd` as a validation to access whatever proof is eventually supplied for this. Low-level tactics may be defined in this way, from first principles, by using the `refine` function to access primitive rules. Tactics may also be defined by composing previously defined tactics using higher-order functions called *tacticals*, with the advantage that tactic libraries may be built up in a modular, re-usable fashion. Tacticals typically take tactics (and possibly some additional arguments) and return a tactic. Tactics built in this way often have explicitly recursive (i.e. `letrec`) definitions or are defined in terms of recursive tacticals such as `REPEAT` below and its relatives recursion is typically on subgoals and subterms. The failure mechanism is also made extensive use of in this context. Some of the more common tacticals are[3]:

`THEN : tactic -> tactic -> tactic` This is used in infix form to compose two tactics. In applying `f THEN g`, first `f` is applied. If it fails then the whole fails. If not, `g` is applied to the resulting subgoals.

`ORELSE : tactic -> tactic -> tactic` This allows alternation of tactics. In applying `f ORELSE g`, `f` is first applied. If this succeeds then the whole succeeds with the same result. Otherwise the result is that of applying `g`.

---

[3]These aren't quite the definitions used in NuPRL, where non-progress is treated as failure. We shall overlook this and use the definitions given.

`TRY: tactic -> tactic` In applying `TRY f`, `f` is applied. If this succeeds the whole succeeds with the same result. Otherwise the result is the original goal. This is equivalent to `f ORELSE IDTAC`.

`REPEAT : tactic -> tactic` `REPEAT` repeatedly applies its argument until it fails. It can be defined in terms of the tacticals above as:

```
letrec REPEAT tac = (tac THEN (REPEAT tac)) ORELSE IDTAC;;
```

The termination of `REPEAT f` requires that `f` fail at some point. Thus its totality is not implied by the totality of `f`. This presents us with a problem since it contradicts the totality dictum of section 2.1. We show in chapter 4 how we overcome this by using the Prolog/ML level and the reflection mechanism simultaneously, confining, as it were, all non-total behaviour to the former.

Although any function of the right type may be classed as a tactic, is it necessarily useful? (Gordon *et al*, 1977b) define a tactic as *valid* if its validation does indeed map `shots` achieving the subgoals to a `shot` achieving the goal. This is the minimum requirement for a tactic to be considered useful. In (Paulson, 1983b) a tactic is described as *conservative* if its subgoals are achievable whenever its input goal is. Whereas validity is a property of how an inference is implemented, conservativeness is a property of the inference it itself. It is not always desirable; for example, rules like $\forall$- and $\exists$-intro which require that a choice be made are usually not conservative. (Gordon *et al*, 1977b) define a tactic as *strongly valid* when it is both valid and conservative. We shall see in chapter 3 that our tactics are, if we regard their *preconditions* (q.v.) as subgoals, always valid. They are not in general conservative, but since the precondition will usually be decidable, and therefore treated more like a side condition, they are as good as.

Similarly, a tactical is said to be valid (conservative) when it preserves the validity (conservativeness) of its arguments. Validity is again a minimum requirement to be considered useful. All the tacticals we discuss will be valid and conservative.

Many theorem provers, and of especial interest to us, *CLAM*, rely heavily on rewriting in the course of a proof. (Paulson, 1983a) shows how one can construct

a library of rewriting functions in a manner analogous to that described previously for ordinary tactics. He defines *conversions* as:

```
lettype conv = term -> (term # tactic);;
```

The idea is that a subterm, t, of the goal, $\mathcal{H} \vdash \mathcal{G}$, we wish to rewrite is mapped by a conversion to a pair (u, tac), where *tac should* prove the goal $\mathcal{H} \vdash t = u$ in T (or $\mathcal{H} \vdash t \leftrightarrow u$ where appropriate). This in turn can be used to perform the rewriting using the primitive substitution rule (or simple logical reasoning). Paulson shows how conversions can be combined using analogues of the traditional tacticals. This is of particular interest to us since we shall later limit ourselves (see chapter 3) to synthesizing rewrite tactics.

Most of what was said above holds in the case of the OYSTER re-implementation of NuPRL, translating ML functions to Prolog predicates (Horn, 1988). Proofs are not passed explicitly as input and output by predicates; instead proof occurs as a side-effect of execution, the interface provided by a fixed set of interface predicates. Principal among these are `goal/1`, which returns the conclusion of the current goal, `hyp_list/1`, which returns the hypothesis list of the current goal, and `apply/1`, which applies a refinement to the current goal. The use here of side-effects does not contradict our assertion above that the tactic language should be side-effect-free: for the purpose of reasoning about them the OYSTER tactics and tacticals behave like functions since they are only ever applied via `apply/1` which behaves like a functional interpreter.

To summarize: it appears that in order to specify tactics we shall need: a typed metalanguage, functional rather than procedural, allowing higher-order functions, possessing a mechanism for handling failure, supporting an abstract representation of the object logic, and with good support for recursion.

To avoid later confusion we note here that the word "tactic" is used in three distinct senses in this thesis. The traditional sense is that described above (e.g. particular programs of the OYSTER metalanguage) and to avoid confusion will sometimes be written "OYSTER tactic" etc. We also refer to the functions (terms of the OYSTER

logic) we synthesize as tactics; for clarity, "synthesized tactics". The third type of tactic is described in chapter 5 and is called throughout a pseudo-tactic.

2.3 Methods of specification

There are several different methodologies being used at present for formal program specification. There are also several ways of classifying these, e.g. procedural vs data abstraction, or model- vs property-oriented. Several classes take for granted the use of traditional *procedural* programming languages and are thus poor choices for specifying tactics. We shall include in the survey below so-called declarative languages which are in a sense executable specifications. We do not attempt broad categorizations, instead merely describing and appraising each approach according to: its aims and methodology, and its advantages and disadvantages, from the point of view of tactic synthesis.

Methods of specification and programming language semantics are necessarily closely related. Both attempt to supply meaning, but whereas with a semantics the intention is to give a precise meaning to a language, to all its constructs and syntactic categories, specifications are intended to give perspicuous meaning to a particular instance of the syntactic category *programs*. However, the distinction is blurred. An example where the same notion can be used for both is the use of Hoare-style logic as both a specification language and as the means of giving the programming language semantics (Hoare & Wirth, 1973).

It should be remembered that most of the methods given below were originally intended for *verifying existing* programs or circumscribing implementational decisions so that only correct programs result. With a few exceptions they were also intended primarily for human use. In these two respects they are not ideal tools with which to carry out automated program *synthesis* such use gives rise to large, non goal-directed search spaces. Another problem, common to the automation of any approach, is the perennial trade-off between the expressiveness and the tractability of the formalism used a good example is the complexity

of unification algorithms, where first-order unification is of limited expressiveness but has linear complexity and a most general unifier, in contrast to higher-order unification, which although very expressive is undecidable and may result in an infinite set of independent unifiers (Huet, 1977). A partial answer to both these problems is to direct search using meta-level planning and guidance. This will be covered in later chapters.

Below we list five existing methods by which specifications are given and programs satisfying a specification developed, using a common example to give the flavour of each. We use the factorial function as our example since it is well-known and, like tactics, is side-effect-free and recursive. Some of the methods necessarily overlap

Z evolved from extensive use of the Floyd-Hoare method, for example, and the Curry-Howard isomorphism can be seen as the conjunction of both types of declarative programming but we shall treat each distinctly as do most sources. For a more extensive survey see, e.g., (Harel, 1980).

In (Sannella, 1988), Sannella lists some questions which one should be able to answer for any proposed specification method. These are oriented towards methods which rely on stepwise refinement, but are useful more generally. We list these here, modified slightly for our needs:

1. What is the specification language (SL)?

2. What specification-building (horizontal composition) operations are available in SL?

3. What is the programming language (PL)?

4. What is the relationship between PL and SL? In particular, is there a reflection mechanism?

5. What does "refinement" (vertical composition) mean and under what circumstances is a refinement step correct?

6. How is the transition between SL and PL made?

7. What is the relationship between refinement and decomposition?

8. Does refinement of programs $P \rightsquigarrow P'$ make sense? What is the relation between this and refinement of specifications $SP \rightsquigarrow SP'$?

9. Does the refinement process itself have the status of a formal object subject to analysis and manipulation?

Sannella also gives a list of questions pertinent to the partial automation of the program development process:

1. What *methods* are available for proving the correctness of the refinement of steps?

2. Where do refinement steps come from?

3. What *tools* are available for assisting with which aspects of program development?

4. What level of sophistication is required of the user of such tools?

5. Which aspects of the program development process can be fully automated?

6. Does the approach require specifications to be "complete" in any sense? Does it provide a way of checking completeness of a specification or identifying areas of incompleteness?

7. Does the approach provide ways of deriving programs which are optimal (or at least adequate) with respect to some performance measure?

8. Does the approach provide a formal way of comparing pros and cons of different implementations which meet a specification.

9. What are the complexity properties of the approach; for example, what happens to the size of proofs of correctness as specifications grow in size?

We have attempted to answer these questions for each of the methodologies below.

2.3.1 Floyd-Hoare style specification

The Floyd-Hoare style (Floyd, 1967; Hoare, 1969) of specification typifies the input/output specification paradigm. This relies on there being an execution state and a flow of control which moves around a program's source code as execution proceeds. In (Floyd, 1967) this was a flowchart; (Hoare, 1969) formalized the idea by applying it directly to a program's abstract syntax. For this reason the Floyd-Hoare method is often classed as *syntax-directed* (vs *data-directed*, see section 2.3.3). Specification consists in writing an assertion of the form:

$$\{P\}Prog\{Q\}$$

This has the meaning "if proposition P is true of the state when control enters the (sub)program $Prog$, then, upon termination, Q is true of the resulting state." The state is referred to by identifying the program variables and logical variables. P and Q are assertions in a suitable logic, e.g. first-order predicate logic, allowing one to reason in the normal way about relationships between the before and after states. Due to the "upon termination" clause, this form of assertion specifies only *partial* correctness. Termination is usually proved separately by considering for each loop in the program some quantity, expressed in terms of the program variables, which is shown to decrease in some well-founded order on each iteration (Floyd, 1967). For this reason a stronger theory is used to account for reasoning about the specific data types over which a program operates, e.g. PA for natural numbers. There are Hoare logics for specifying total correctness but these are much more complicated, typically requiring second order quantification (Apt, 1978).

Each programming construct has a corresponding inference rule in the logic. Figures 2 1 and 2 2 give those for assignment and `while`. The consequence rule (figure 2 3) is used to link deductions in the underlying logic to those in the Hoare logic. Each programming construct gives rise to a verification condition: the program is verified (partially correct) when the verification conditions for each construct in it have been proved in the logic. For example the verification condition for $\{P\}\mathtt{x:=t}\{Q\}$ is $P[t/x] \rightarrow Q$. This form of specification, for imperative languages at least, is still the most widespread in real world use (Goguen, 1986). It usu-

$$\frac{}{\{\mathcal{P}[t/x]\}\mathtt{x:=t}\{\mathcal{P}\}}$$

Figure 2–1: The assignment rule

$$\frac{\{\mathcal{I} \wedge \mathtt{B}\}\mathtt{L}\{\mathcal{I}\}}{\{\mathcal{I}\}\mathtt{while\ B\ do\ L}\{\mathcal{I} \wedge \neg \mathtt{B}\}}$$

Figure 2–2: The `while` rule

ally appears in a much extended form, such as Z (see below), or a special-purpose programming logic (Harel, 1984; Apt, 1978) of which an early example is the *intermittent assertion* method of Burstall (Burstall, 1974; Manna & Waldinger, 1978)

Floyd, in his early paper (Floyd, 1967), hints at the possibility of automating much of the work of program *verification*. This involves the programmer in providing an overall input/output specification as well as (at least) one *invariant assertion* for each cycle in the program flowchart. An invariant is a property that is preserved between the input and output states by a program. Discovering a suitable invariant typically requires some insight into how the program works[4]. $\mathcal{I}$ is an invariant of L in figure 2 2. A theorem prover would then be used to show that *verification conditions* hold between adjacent sub-programs. The overall input/output specification is propagated throughout the program by means of weakest pre-condition/strongest-post conditions (de Bakker, 1980), and inside loops via the given loop invariants. An implementation of such a system is given in (Gordon, 1988).

[4]It is theoretically impossible to generate sufficient loop invariants automatically, since this would allow one to solve the halting problem (Harel, 1988). However, the process can be *partially* automated (Chadha & Plaisted, 1993), completely so for theories with finite first-order axiomatizations (not PA).

$$\frac{\mathcal{P} \rightarrow \mathcal{Q} \qquad \{\mathcal{Q}\}\mathtt{L}\{\mathcal{R}\} \qquad \mathcal{Q} \rightarrow \mathcal{S}}{\{\mathcal{P}\}\mathtt{L}\{\mathcal{S}\}\mathtt{B}\}}$$

Figure 2–3: The consequence rule

We give an example of how one would specify a program, $Prog$, to calculate the factorial (in the variable `y`) of the input variable `x`. Variables are assumed to range over the natural numbers and the theory we use is Peano arithmetic.

```
y:=1;
while (x>0) do
 y:=y*x;
 x:=x-1
od
```

One would give the specification $\{x = a\}Prog\{y = a!\}$. This uses the common trick of connecting the before and after states through variables (here a) which don't appear in the program. The program above can be verified using the three Hoare rules given. Below we have used the loop invariant $x!y = a!$. The same program through which the overall specification and loop invariant have been propagated looks like:

$$\{x = a\}$$
`y:=1;`
$$\{x = a \wedge y = 1\}$$
$$\{x!y = a!\}$$
`while (x>0) do`
$$\{x!y = a! \wedge x > 0\}$$
`y:=y*x;`
$$\{x!y = a!x \wedge x > 0\}$$
`x:=x-1`
$$\{(x+1)!y = a!(x+1)\}$$
$$\{x!y = a!\}$$
`od`
$$\{x!y = a! \wedge x \leq 0\}$$
$$\{y = a!\}$$

We could prove termination by showing that the value of x strictly decreases each time execution passes round the loop `y:=y*x ; x:=x-1`. Since the natural numbers are well-founded, this process must eventually terminate. We have thus shown that the program is totally correct with respect to the given pre- and post-conditions.

Despite its widespread use the method has several drawbacks which we quote from (Goguen, 1986):

- Expressing complex specifications in first-order logic, or any of the usual simple logical languages, yields very complex sentences, which are consequently often wrong.
- It is impossible to give a complete set of Hoare-style rules for programming languages as rich as the ones usually used in computer science.
- We do not know how to give simple Hoare-style rules for some common features of today's programming languages.
- It is, in general, necessary for the user to supply so-called invariants in order to prove the correctness of programs with iteration or recursion.
- It is often very difficult to prove the hypotheses which are generated by the verification condition generator, and consequently it is very difficult to believe any proofs which may be offered.
- In order for this standard paradigm to be rigorously correct for a given programming language, one should have a formal definition and a correctness proof for the verification condition generator used for that language.
- One should also have some independent check of the correctness of the Hoare-style rules, such as a proof that they are valid in some model, and are therefore also consistent.

The second and third points above are made explicit in, respectively, (Clarke, 1979; O'Donnell, 1982). The former shows that it is impossible to obtain a sound and

complete (in the sense of (Cook, 1978)) system of Hoare axioms for a programming language which allows: procedures as parameters of procedure calls, recursion, static scoping, global variables and internal procedures. As the introductory discussion makes clear, these are all desirable features of a tactic programming language. Since soundness and completeness are minimal requirements for a "good" specification language this presents a serious obstacle. The second paper, though not providing a non-existence proof, indicates that a simple and elegant Hoare rule for a construct as simple as function definition appears unlikely. This is due to the fact that defined functions bring together the previously independent notions of partial correctness and termination in such a way that is no longer convenient to separate them, with the conclusion that for reasoning about defined functions total correctness logics are best used. The inference from these papers is that while Hoare logic is useful and elegant for reasoning about simple languages, its extension to the features found in modern programming languages results in unnatural rules which are difficult to reason with.

Since these criticisms apply a fortiori to tactic programming languages we feel that Floyd-Hoare style specification, and the input/output paradigm in general, to be ill-suited to specifying tactics. The case is even worse when one considers program synthesis: being syntax-directed is a serious disadvantage when talking about a non-existent program. Hoare-style specifications resulting in very large and poorly-directed search.

2.3.2 Z

The Z specification language (Spivey, 1988b), similar in style to the Vienna Development Method (VDM), uses a weakly typed version of Zermelo-Fraenkel set theory to specify program behaviour. The intention with Z is to allow the flexible expression of extremely abstract and high-level initial specifications and to then introduce successive refinements until, eventually, a machine-implementable form is reached, e.g. a program in a traditional programming language. In this way a Z

specification provides an "intellectual handle" for a programmer or programming team.

However, unlike the other means of specification described in this chapter, Z is less concerned with providing a particular mathematical foundation in which to set program specification than with laying down a set of rules for consistently using such a foundation. Thus, Z is more methodology than method.

A Z specification takes the form of a linear script. The principal construct of Z is the *schema*, a declaration constrained by a predicate, which has the general form

$$Name[X_1, \ldots, X_m] \widehat{=} a_1 : T_1, \ldots, a_n : T_n \bullet Pred$$

The X_i are generic variables (variables which will be instantiated with sets) and the T_j are types (sets of a particular form) given in terms of these, primitive types and types defined previously. $Pred$ is a formula of the first order theory of ZF all of whose free variables are amongst the X_i and a_j. This should be read as: "given sets $X_1, \ldots, X_m$, there exist respectively elements $a_1, \ldots, a_n$ of type $T_1, \ldots, T_n$ such that $Pred$ is true." Thus a schema corresponds to the more usual notion of module. Spivey provides a metacircular semantics for Z in (Spivey, 1988a) a very basic, more obviously acceptable, version of Z is used to specify the full-blown Z. A powerful feature of Z is the *schema calculus* which allows schemas to be combined to form new schemas. A typical example is the forming of a robust specification from those for normal and erroneous operation:

$$RobustOp \widehat{=} (NormalOp \wedge OK) \vee Error$$

Here the declarations of all the schemas are combined in the new schema and the predicates are joined using the logical connectives shown. Note that no direction is given as to how variables are to behave in the new schema, for example whether identical variables are to be renamed.

Because of the intended use of Z mentioned above, our common example is not a good one in this case. Nevertheless:

$$Fact \widehat{=} fac : N \rightarrow N \bullet fac(0) = 1 \wedge \forall x : N . fac(x+1) = (x+1) * fac(x)$$

This schema might appear as a single line in a much larger Z specification. It states that a function called fac satisfies the properties given.

It is precisely the reasons that make Z a good specification language for human use that make it ill-suited for use in automatic program synthesis. Specifically:

- Z's very abstract and conceptual approach provides poor guidance for a theorem prover. For example, functions and relations are given in terms of their graphs, and the weak typing available does not explicitly distinguish between various forms of these.

- Z is inherently biased towards procedural programming languages. Iteration is the usual form of looping. In implementing tactics we are almost exclusively interested in recursion.

- To ease use, the notation involves much implicit and hidden information. For example, the typing used is very weak, most of the notation adding implicitly to the predicate part of a schema. There is also the convention of decorating variables according to their use, e.g. $x?$, $y!$, z, z' for respectively input, output, before and after. This is a form of binding quite alien to traditional logic.

- There is no formal link, as for instance there is in the case of the Curry-Howard isomorphism, between a Z specification and an implementation. The final refinement may be so concrete that the required program fragment is obvious. However, to be completely formal we would have to fully specify some programming language in Z and take it on trust that this was satisfied by the implementation used.

- As mentioned above the schema calculus depends on a common understanding of bound variables depending purely on name. Thus, there is an uneasy mix between syntax and semantics. This leads to instances where referential transparency breaks down. Such behaviour is particularly undesirable in the context of theorem proving where one expects the usual properties of equality to hold.

- In addition to the above, there are also problems with the semantics of Z which are examined in (Spivey, 1988a).

For these reasons we feel that Z is unsuitable for the task of specifying tactics.[5]

2.3.3 Algebraic specification

The proponents of algebraic specification argue the primacy of the basic data types employed when constructing a program. The choice of these types and the operations defined upon them crucially affects the later design stages, even the algorithmic structure of a program. For this reason algebraic specification is classed as a *data-directed* method. For a thorough treatment we refer the reader to (Wirsing, 1990; Ehrig & Mahr, 1985).

An algebraic specification consists of the description of one or more *abstract data types* by giving the names of sorts and the names and characteristics of the basic functions defined on the sorts. The first two comprise the *signature* of a specification and the last the *axioms*. From these can be determined the semantics of a specification. The semantics is described in terms of those structures which are models of the specification, i.e. the many-sorted algebras (in the usual mathematical sense) which satisfy the specification[6]. There are three main approaches to providing the semantics. The *loose* approach considers all possible models (analogous to the model theory of validity in FOPL). This can be viewed as working with an unfinished specification which is tightened up as more axioms are added. One has achieved a complete specification when only one model remains — a *monomorphic* specification. A second approach is to consider a particular model called the *initial algebra*, where it exists. This has the appeal of corresponding to the intuitive notion of the quotient of the term algebra (see e.g. (Ehrig & Mahr, 1985)).

---

[5]Tactic specification via Z has, however, been attempted elsewhere: (?, 1992).

[6]In fact, the models are taken to be the isomorphism classes of these algebras. This is indicated by the use of the prefix *abstract*.

Such an approach is desirable when one wishes to associate with each specification an explicit model. Initial algebra specifications also have the advantage that they result in true (conditional) equational specifications in situations where the loose approach requires inequalities (see the example below). A third approach is the dual of the previous: use of the *terminal algebra*. The initial (respectively terminal) algebra semantics can be characterized as the model in which only those objects are equal (respectively distinct) which can be proved equal (respectively distinct) by the axioms.

Although no mention was made above of how the characteristics of an operation are described, most work to date has been concentrated in the setting of first-order equations or conditional equations. This restriction results in many desirable properties (e.g. sound and complete deductive systems), but at the same time hinders the writing of more natural specifications: an example of the trade-off between expressiveness and tractability.

In the style of (Wirsing, 1990) our common example becomes:

spec NAT ≡
  signature
    sort nat
    functions
      0 : → nat
      fact, succ : nat → nat
      +, * : nat,nat → nat
  endsig
  axioms
    succ(x)=succ(y) → x=y
    succ(x) ≠ 0
    0+y=y
    succ(x)+y=succ(x+y)
    0*y=0
    succ(x)*y=y+(x*y)
    fact(0)=succ(0)

fact(succ(x))=succ(x)*fact(x)

endspec

The loose semantics of this specification admits non-standard models as well as the standard model of arithmetic. By considering only *generated* or no "junk" algebras, i.e. those where all elements of the carrier sets denote some ground term, the same specification becomes monomorphic, having as its sole model the standard one. However, it will be noted that this is still at the expense of using an inequality. By considering only the initial algebras, we may drop the first two axioms, giving a purely equational specification.

Besides allowing us to analyse data types in the ways outlined above, algebraic specification methods are largely concerned with providing for *structured specification* and *validation*. The former will be of little interest to us, allowing as it does the building of *hierarchical* and *parameterized* specifications: the tactics we shall be considering are not large enough to warrant the use of such techniques. Validation is a point touched on in each of these sections. Formally validated tactics, via some form of reflection mechanism, may be used directly, obviating the need for full execution of a tactic or compilation to a separate tactic language (see chapter 3) (Goguen & Winkler, 1988). Validation is carried out in one of two ways: use of a specification language which supports direct interpretation/compilation of a specification; or the technique of successive refinement, where at each stage a specification is implemented in terms of a more concrete one, eventually arriving at an executable specification.

From this brief description it will be seen that the algebraic method of specification is not well suited to our needs:

- The emphasis is wrong: algebraic specification is data- and model-oriented. We shall be using a small number of given data types which, in themselves, are of no interest. Similarly, we are not interested in how loose a specification is but in finding a program that satisfies this.

- Most current techniques are limited to first-order conditional equational specifications. Whilst this is theoretically sufficiently expressive, it is awkward in practice. An example is the problem of specifying erroneous operations (Goguen *et al*, 1978), where changes made to one sort must propagated across the whole specification. We also need a higher-order language to specify tacticals.

- While the reasoning used in tactic synthesis will in the main part be algebraic, i.e. consist of the use of first- and higher-order conditional rewrite rules, the way in which such reasoning is directed does not follow the refinement paradigm mentioned above.

For these reasons we choose not to use typical algebraic specification methods for tactic synthesis. However, all syntheses will use much algebraic reasoning and this aspect will be subsumed by the method we eventually choose.

2.3.4 Declarative languages

"Declarative" is the term coined to describe programming languages whose programs in some sense "declare" their meanings. Rather than describe exactly how to perform a calculation, as in the While program above for calculating the factorial, programs simply specify the properties of a function or relation. It is up to the language designer to make such specifications executable. There are two main camps in declarative programming: logic (or relational) programming and functional (or equational) programming. An example of the former is Prolog, where a program consists of logical assertions about various predicates. ML, where functions are specified by equations between terms of the simply-typed λ-calculus, is an example of the latter.

In order to keep things tractable, programs are restricted to forms to which a procedural interpretation can be given: Horn clauses in the case of Prolog, where the interpretation is SLDNF resolution (see, e.g., (Lloyd, 1987)); first-order patterns for the left-hand sides of equations and a static typing regime in the case of ML.

where the interpretation is call-by-value reduction. In idealized versions of these languages (logic programming and functional programming respectively) there is a simple mathematical semantics, viz., for Prolog query Q and ML ground term T of base type:

$$\begin{aligned} Q \text{ succeeds with instantiation } \theta &\implies Comp(Prog) \models Q\theta \\ Q \text{ succeeds with instantiation } \theta' \leq \theta &\impliedby Comp(Prog) \models Q\theta \\ T \text{ evaluates to } t &\implies Prog \vdash_{\lambda\beta_v} T = t \\ T \text{ evaluates to } t &\impliedby Prog \vdash_{\lambda\beta_v} T = t \text{ and } t \text{ is normal} \end{aligned}$$

Here $\models$ is the usual semantic turnstile of first-order predicate logic, $Comp(Prog)$ refers to the Clark completion of a logic program (Clark, 1978), and $\theta' \leq \theta$ means that θ is an instantiation of θ'; $\vdash_{\lambda\beta_v}$ refers to the theory of typed β-equality (see (Hindley & Seldin, 1986)). In practice, only the left-to-right implications (soundness) hold since programs in either language need not necessarily terminate. The right-to-left implications hold if one employs a "fair" notion of execution in the case of Prolog, one where all the non-deterministic paths in an execution have the same precedence, and in the ML if the evaluation terminates. We have considered only a very small subset of ML here, ignoring exceptions, references, assignments, and even simple features such as type constructors other than $\rightarrow$. We have similarly not considered the non-logical features of Prolog such as input/out and use of the cut (!) operator. As one takes into account more of the practical limitations of a declarative language, its semantics becomes more complex accordingly.

We continue with the common example, first in Prolog:

```
fac(0,s(0)).
fac(s(N),X):-fac(N,XX),times(s(N),XX,X).
```

Assuming that we have a simple theory $PNat$ for the naturals and the predicates `plus` and `times`, we can prove $comp(PNat + FacProg) \vdash fac(X, X!)$, where we may define the factorial function ! in any manner convenient in the logic. Since $\vdash$, the proof-theoretic turnstile for the first-order predicate calculus, is sound for $\models$, we will have shown that (at least one of) the answer(s) to such a query is correct.

In ML we have:

```
let fac(0) = s(0) |
    fac(s(n)) = s(n)*fac(n);;
```

Under the similar assumptions we can prove $PNat' + FacProg' \vdash_{\lambda\beta_t} fac(n) = n!$.

Using a declarative language for tactic specification seems very attractive:

- These are the languages that present-day tactics are written in. The two mentioned, Prolog and ML, have explicit notions of failure.
- These programs are, in comparison to procedural languages, easy to reason about. We can use traditional theorem proving techniques.
- We have available a reflection mechanism in the form of meta-programming (Giunchiglia & Traverso, 1990).

However, there are some problems. These languages are general purpose and programs written in them are not inherently total. At the same time the restrictions used to give a procedural interpretation limit expressiveness — it is difficult to express negative and disjunctive information in both Prolog and ML. It is also not obvious at what point a declarative program becomes executable. By using the method of the next section we hope to retain the good points of a declarative language whilst avoiding the bad.

2.3.5 The Curry-Howard Isomorphism

The Curry-Howard isomorphism gives an exact correspondence between the logical and functional aspects of computation — a proposition is true iff it is inhabited; a term is a proof of the proposition it inhabits. In analogy with the Floyd-Hoare method one could say that a proposition is like an invariant of the inhabiting term and that the type regime ensures termination. Heyting (Heyting, 1956) and Kolmogorov (Kolmogorov, 1932) were the first to note the

isomorphism, Howard (Howard, 1980) made it precise, and today it is implemented in various intuitionistic type theories (Martin-Löf, 1984; Constable *et al*, 1986; Coquand & Huet, 1988; J.-Y. Girard & Taylor, 1989). More precisely, we have the correspondence:

Proposition	Type	Canonical inhabitant(s)
$false$	void	
$\mathcal{A} \wedge \mathcal{B}$	$\mathcal{A}\#\mathcal{B}$	$< a, b >$
$\mathcal{A} \Rightarrow \mathcal{B}$	$\mathcal{A} \rightarrow \mathcal{B}$	$\lambda x.b$
$\mathcal{A} \vee \mathcal{B}$	$\mathcal{A}\|\mathcal{B}$	$\mathtt{inl}(a)$ or $\mathtt{inr}(b)$
$\forall x : \mathcal{A}.\mathcal{B}(x)$	$\Pi x : \mathcal{A}.\mathcal{B}(x)$	$\lambda x.b$
$\exists x : \mathcal{A}.\mathcal{B}(x)$	$\Sigma x : \mathcal{A}.\mathcal{B}(x)$	$< x, b >$

In proving a theorem in type theory we implicitly construct a λ-calculus term (in NuPRL this is untyped, in the Calculus of Constructions (CC) typed). It is these terms which are both the elements of the types and constitute the programming language of type theory. For example, by proving a theorem of the form $\forall x : A.\exists y : B.spec(x, y)$ in one of the type theories mentioned, we can extract from it a term t such that for any canonical term a of type A, $t(a)$ reduces to a term $< prog(a), P >$, where P is a proof that $spec(a, prog(a))$. Execution in the lambda calculus is via a given reduction strategy , and all well-typed terms (which includes the extracts of proved theorems) normalize to a canonical form. Thus, we are guaranteed total programs. This is how synthesis is achieved using the Curry-Howard isomorphism. Conversely, we can accomplish verification by showing that $\forall x : A.spec(x, prog(x))$.

Of special interest to us is the way recursion arises in type theory. Each primitive, recursive type, e.g. `pnat`, A `list`, has an associated elimination rule. This is what is more commonly known as the primitive induction rule for that type. For `pnat`this is:

$$\frac{x : \mathtt{pnat} \vdash T[0/x] \qquad x : \mathtt{pnat}, p : \mathtt{pnat}, v : T[p/x] \vdash T[\mathtt{s}(p)/x]}{x : \mathtt{pnat} \vdash T}\ \mathrm{elim}(x)$$

The extract from such an inference step is $\texttt{p\_ind}(x; b; p, v.s)$ where b and s are respectively the extracts from the base and step cases. Although such recursion is always primitive, it can take place at any type T; so, for instance, it is easy to define Ackermann's function by primitive recursion at a functional type. User-defined recursive types (`rec`types) have similar induction rules and recursion constructors. In this way there is a duality between induction schemas and the recursion schemas of their extracts. In addition, since NuPRL has type universes, one is able to prove and use as lemmas customized induction schemas rather than repeatedly proving from first principles.

We now apply the Curry-Howard synthesis paradigm to our example. By proving the theorem

$$\vdash \forall x : \texttt{pnat}.\exists y : \texttt{pnat}.\texttt{factorialp}(x, y),$$

where `factorialp` is axiomatized as in the Prolog example above, we are able to extract the program

$$\texttt{prog} = \lambda x.\texttt{p\_ind}(x; < s(0), \ldots >; v0, v1.\texttt{spread}(v1; y, v2. < \texttt{times}(\texttt{s}(v0), y), \ldots >))$$

and so set

$$\texttt{fac} = \lambda n.\texttt{fst}\,(\texttt{prog}\; n).$$

There are three other advantages which follow directly from the use of type theory. Firstly, synthesis *is* theorem proving. Since we are dealing with propositions in a logical language there is a greater degree of goal-directedness with this method than with any of those above. This allows us to apply a large number of existing techniques to this apparently novel domain; in particular, we shall be using the proof planner CL^AM. Secondly, as the work of (Howe, 1988a) shows, we can implement an elegant partial reflection mechanism. Third is the self-reference: we are synthesizing tactics to prove theorems in the system which we are using for the synthesis. In this way we can "bootstrap" our theorem prover.

As can be seen from the simple example above, the program extracted in general contains much *computationally redundant* structure signified there by "..." and requiring the use of the projection function `fst` in the resulting program. Much

work has been done on avoiding this redundancy. This includes the use of subset types (Constable *et al*, 1986), the acc type (Nordström & Smith, 1990) and notions of realizability (Paulin-Mohring, 1989). Deliverables (Burstall & McKinna, 1991) start from the idea that a program and its correctness proof should always be kept apart. More will be said of this problem in chapter 5. There is also the more general problem that having tactics written in the language of type theory means that execution is at least one level of interpretation removed from that at which tactics are traditionally run. However, it should be noted that the slow-down is by a constant factor, there is nothing inherently inefficient about type theoretic languages, and that such trade-offs are usually when choosing a formal language over a machine-oriented one.

Work has previously been done on using type theory to specify tactics. By using a partial reflection mechanism and developing a meta-theory of tactics (see chapter 3) Howe (Howe, 1988a, chap. 5) was able to use NuPRL to verify tactics which could then be used in proving theorems in NuPRL itself. We discuss this and other reflection mechanisms in chapter 6.

For the stated reasons, we have chosen to specify tactics in the type theory of OYSTER.

2.4 Conclusions

The objective of this work is to automate the synthesis of tactics. Thus, besides criteria such as naturalness, expressiveness, flexibility and tractability which one usually considers when looking at methods for specifying and verifying programs, one has additionally to consider the question of how amenable a particular method is to automation, especially with regard to synthesis.

From this perspective we believe that the best medium in which to carry out tactic synthesis is constructive type theory, for the following reasons:

Subsumption Type theory is seen by many (Martin-Löf, 1984; Constable *et al*, 1986; Coquand & Huet, 1988) as a foundation for constructive mathematics. Its open-endedness allows it be used relatively naturally as a medium in which to build other formalisms. Thus, type theory can subsume many of the concepts and formalisms of differing methods (Kreitz, 1993), whereas the converse is not true.

Constructivity We needn't be concerned about explicitly writing a tactic. By proving that a specification formula is true we implicitly construct a (possibly very inefficient) tactic. If necessary, the proof can later be transformed to render it more efficient (Madden, 1991; Goad, 1980).

Totality In a constructive type theory all extracts from completed proofs terminate under normal-order reduction. Thus, we are guaranteed total programs[7]. (We distinguish non-termination from failure, a mechanism for which is easily developed.) This maybe seen as a weakness of choosing type theory since one may want to define a tactic for a generally undecidable problem, typically one involving search. As mentioned in the introduction, however, we are content to restrict tactics to being total.

Automation In synthesizing a function in type theory we prove the corresponding specification theorem. So already we are proving a theorem in a familiar logic, where the usual rules of inference apply, instead of, say, trying to first synthesize and then verify a program using Hoare-logic, where things such as assignment must be reasoned about.

Induction The recursive structure of tactics will be of prime importance in guiding synthesis proofs. Type theory makes any induction and its dual recursion explicit.

One should not pretend that setting our specifications in type theory is a panacea for the problems of program synthesis. It does give us a precise formalism,

---

[7]In fact, this is the starting point of constructive type theory (Martin-Löf, 1979).

clear semantics and flexible medium with which to carry out tactic synthesis. We summarize our decision below.

The foregoing survey convinced us that the tool best-suited to our needs was the Curry-Howard isomorphism in the form of an intuitionistic type theory such as OYSTER. We can characterize the various methods according to: to what extent their intended usage matches ours; their means of and usefulness for program synthesis; their amenability to automation and existing tools for them, such as theorem provers; and the possibility of using a reflection mechanism. The results are summarized below.

The Floyd-Hoare methodology, due mainly to its concentration on the assignment rule, is limited to work with imperative languages and hence unsuitable for work with the tactics we are interested in. It also constitutes a very poor candidate for program synthesis which takes the form of eureka steps plus post hoc verification. Although some work on automating the production of verification conditions and loop invariants has been carried out (Gordon, 1988) there are few software tools available for work in this area. It is hard to see how a reflection mechanism could be naturally constructed in this setting.

Z is intended to formalize the design and implementation of large software projects. Its bias towards human perspicuity is clear in its large and rich number of constructs. Synthesis is limited to stepwise refinement, making it a poor candidate for program synthesis. However, there is a large array of software tools for carrying out verification proofs in Z (Jones, 1992; Bowen & Gordon, 1994). The author is aware of (at least) one attempt at constructing a reflection mechanism in Z.

Algebraic specification is useful in bringing a rigorous semantics and structure-enforcing framework to functional programming. However, in its usual form its restriction to plain or simple conditional equations can lead to somewhat convoluted specifications. Offering stepwise refinement and direct interpretation as methods for program synthesis, it is a relatively good candidate for program synthesis work. Since reasoning is for the most part equational, automation is promising and there are a number of systems in existence (Goguen, 1989; Futatsugi *et al.* 1985;

Burstall & Goguen, 1977). One of these systems includes a reflection mechanism (Goguen & Winkler, 1988).

Declarative languages too hold promise for work in the area of tactic synthesis. While having a formal semantics, they are more computationally biased and are in general immediately executable, providing a good candidate for program synthesis. The latter achieved via transformation, compilation and meta-programming. There is much existing work on automatic program synthesis and transformation (Boyer & Moore, 1 Wadler, 1990; Burstall & Darlington, 1977) and some on reflection mechanisms (Boyer & Strother Moore, 1981).

Type theory, in its role as a foundational theory, subsumes all of the above (Constable *et al*, 1986; Kreitz, 1993). The Curry-Howard isomorphism provides a direct link between logic (where we may use theorem proving tools such as OYSTER and CL^AM) and computation (programs we may use as tactics), and is therefore an excellent candidate for program synthesis. Work at Cornell has shown how type theory can be used to construct a useful reflection mechanism (Howe, 1988a; Knoblock, 1987).

Chapter 3

Reflection and Representation

3.1 Introduction

In the previous chapter it was argued that the best setting in which to carry out tactic synthesis was constructive type theory, using the Curry-Howard isomorphism to realise actual tactics. It was not made clear, however, exactly how this was to be done. In this chapter we give a concrete formalism for specifying tactics and all the other notions that this requires. These two aspects, the means of specification and the representation used, are almost orthogonal: the representation given below could equally well have been constructed using one of the other specification methods mentioned in the previous chapter.

Guiding the choice of representation has been the possibility of using a reflection mechanism, which would allow us to use the extract of a synthesis proof directly. The kind of tactics we hope to synthesize, those with a cheap meta-level/expensive object-level bias, favour the efficient use of reflection. However, the representation arising from the mechanism is not limited to its use here alone: we shall see in chapter 5 that an extract can be translated into a traditional tactic in a language such as ML or Prolog. Two canonical examples of a reflection mechanism of the type given below are Boyer and Moore's use of meta-functions (Boyer & Strother Moore, 1981), and Howe's partial reflection mechan-

ism (Howe, 1988a). We draw heavily on the latter for the reflection mechanism exhibited below. A comparison with their work is given in chapter 6.

Whenever we wish to study something using a formal system we first have to be able to represent it in that system. So it is with tactics. Tactics are *programs* which construct *proofs*, which themselves contain *formulas* and *terms*. Thus, if we are to reason about tactics we must at least be able to represent such objects. Proofs and formulas are complicated objects, requiring notions of derivability and binding respectively. As Knoblock and Constable show (Knoblock & Constable, 1986), representing the proof theory of a formal system can become very complicated. (This approach was later abandoned due to its impracticability.) Boyer and Moore (Boyer & Strother Moore, 1981) show us that we need not consider the whole system, and may concentrate instead on its underlying notion of meaning; Howe calls this *semantic* reflection. This allows us to use the original proof system to show that certain functions preserve this meaning. Howe (Howe, 1988a) is able to combine this idea with Paulson's of modularly building rewriting tactics (Paulson, 1983a). It is a gross simplification of Howe's work which has been implemented here and is described below.

As our principle interest is the automation aspect of tactic synthesis and not the reflection mechanism itself, we have simplified the mechanism as far as possible. But we have retained enough functionality so that one can see immediately how to scale up our methods for tactic synthesis to a system such as Howe's. The simplifications we have made with respect to Howe's system are that we consider terms of only one type (this could be an arbitrary type but we choose `pnat`), and therefore can dispense with type environments and much of the special machinery for rewriting propositions; and we restrict terms to be at most binary branching (see below).

In the sequel we shall often print nodes from various OYSTER proofs. These are used either to show a proof in progress or, more often, to illustrate the definition of an object as a `term_of` that theorem. The full library listing for the reflection mechanism is given in appendix D. To avoid confusion we use the term "OYSTER

tactic" to mean a tactic in the usual sense as distinct from the objects we wish to synthesize.

3.2 Representing Terms

Like other existing implementations of semantic reflection we shall restrict our attention to the first-order structure of our object language. This means that our synthetic tactics will not be aware of or able to manipulate any binding structures present in a language. However, as the existing implementations show, interesting and useful tactics are still expressible. In general, when we consider first-order terms, we see them as freely generated trees, having variables for leaves and nodes labelled with function symbols and an arity. This is what we mimic in our representation.

Without loss of generality we restrict our term trees to at most binary branching. Whereas a restriction to at most unary branching would have a significant effect (e.g. classical predicate logic restricted to at most unary predicate symbols is decidable (Church, 1951)), we can map, if necessary, an arbitrary term tree to a binary tree. For example, suppose we had the ternary function symbol `f`. We could map the term `f(t1,t2,t3)` to `g(t1*,h(t2*,t3*))`, where `g` and `h` are new binary function symbols. All terms in the image of this map, `t*`, can be mapped uniquely back to the original term.

Restricting our term trees to at most binary branching affords us another advantage besides simplicity. It clearly separates the various cases that occur when we do induction over terms. The variable case is interesting since this is the place at which unification and the like takes place; constants and unary functions introduce the usual base and step cases; and the binary case covers "the rest". As mentioned above, the introduction of binary function symbols constitutes a major step up in computational complexity, and there is little more to be learned from allowing ternary and symbols of greater arity. Thus we can expect the binary case to contain most of the interesting part of any proof where induction over terms is

involved, for example, the joining of two separate unifiers. If we had used a representation employing, for example, lists to give branches of arbitrary arity as in (Howe, 1988a), while admittedly more elegant, the resulting inductions over lists would interfere with and obscure the behaviour of the term inductions.

To represent terms we introduce the inductive type `metaterm`, in `U1`, defined by:

```
metaterm == rec(z.(atom|atom)|((atom#z)|(atom#z#z))).
```

This will be more familiar if rewritten in pseudo-ML code:

```
type metaterm == var of atom | con of atom | una of atom*metaterm
               | bin of atom*metaterm*metaterm;;
```

For ease of use we define the four canonical injections into it:

$$\begin{aligned} \texttt{tvar}(v) &= \texttt{inl}(\texttt{inl}(v)) \\ \texttt{tcon}(c) &= \texttt{inl}(\texttt{inr}(c)) \\ \texttt{tuna}(f,a) &= \texttt{inr}(\texttt{inl}(< f,a >)) \\ \texttt{tbin}(f,a,b) &= \texttt{inr}(\texttt{inr}(< f,< a,b >>)) \end{aligned}$$

To elucidate the idea we give an example. The object level term

`f(x,c)`,

where `c` is a constant and `x` a variable, would have the meta level representation

`inr(inr(<"f",<inl(inl("x")),inl(inr("c"))>)),`

or marginally more perspicuously,

`tbin("f",tvar("x"),tcon("c")).`

We provide a theorem stating primitive induction for `metaterm` in `scheme(metascheme)`.

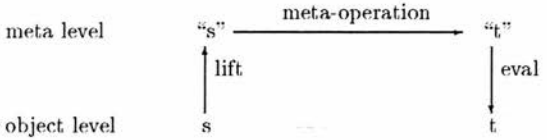

Figure 3–1: The reflection mechanism

3.3 The reflection mechanism

The basic idea is that we demonstrate the soundness of a meta-operation by showing that it preserves equality at the object level. This is done in the following way.

Suppose we have a goal of the form

$$\mathcal{H} \vdash \mathcal{G}[s]$$

where s is a subterm of type `pnat`, and we wish to substitute t for s. At the object level this is justified by a primitive substitution rule which obliges us to show that

$$\ldots \vdash s = t \texttt{ in pnat}$$

Although the form of t may be immediately evident from that of s, for example in a normal forming operation, the proof that both terms are equal may be very expensive to construct. When such an operation is to be performed repeatedly, as it might be by a tactic, we would ordinarily "compile out" these subproofs by proving a lemma once and using this on all further occasions. However, at the object level we are not able to talk about terms intensionally, and so cannot express such "theorems" as:

"a term (s) consisting of a string of + operations = the same term with all +'s associated to the right (t) in `pnat`."

The reflection mechanism allows us to express, prove and use such theorems inside our original system. It does this by allowing us to express and prove the equality

represented by the topmost arrow of fig. 3 1, and by using the proof system's own evaluation mechanism to prove the equalities represented by the two vertical arrows. The transitivity of equality gives us the equality between s and t.

In order to prove the top line we must first give it a meaning and this, along with the vertical lines, is provided by *lifting* and its converse *evaluation*. We lift a term of type `pnat` by constructing a term which represents purely its syntactic form. An example of this was given above for the term `f(x,c)`. We give meaning to the lifted term by supplying an evaluator for metaterms, the function `eval`. Since we may chose to lift a term at an arbitrary point in a proof and in the context of an arbitrary theory, `eval` must be parameterized over both. Context within a proof is given by declarations in the hypothesis list. We expect the meaning of `tvar("x")` to be the variable x and so forth. These variable bindings are provided by a parameter to `eval` of type `env`, i.e. `atom → pnat`. Context within a theory is provided by the definition of functions/constants appearing in a goal. These definitions are supplied to `eval` by a second parameter of type `f_env`, i.e. `(atom → pnat)#(atom → pnat → pnat)#(atom → pnat → pnat → pnat)`, respectively the meanings for nullary, unary and binary function symbols. Hence, `eval` is defined in:

```
eval:  complete
⊢ metaterm → env → f_env → pnat
Extract:  ...
```

The syntactic sugar `eval(m,e,f)` will be used from now on. Since the `env` supplied to `eval` is dependent on the goal we are trying to prove it is built at reflect-time by the predicate `lift_term/4`. This takes an object term, and returns a member of `metaterm` representing it and an accompanying member of `(atom#pnat) list`. The latter is an association list which, along with the OYSTER function `list_to_env` defined in

```
list_to_env:  complete
⊢ (atom#pnat) list → env
Extract:  ...
```

is used to provide the corresponding `env` for `eval` at reflect-time. If `lift_term` arrives at a subterm which it is unable to decompose any further, it is lifted en bloc and given a label of the form `refN` (see example, p. 53).

The `f_env` will usually be constant inside a given theory, so we expect the user to supply this at reflect-time. In the examples below we use the `f_env basic` defined in `basic_f_env`. This supplies the following mappings:

$$
\begin{aligned}
\texttt{``0''} &\mapsto 0 \\
\texttt{``s''} &\mapsto \lambda x.\mathtt{s}(x) \\
\texttt{``+''} &\mapsto \lambda x, y.\mathtt{plus}(x, y) \\
\texttt{``*''} &\mapsto \lambda x, y.\mathtt{times}(x, y) \\
\texttt{``wh''} &\mapsto \lambda x.x \\
\texttt{``wf''} &\mapsto \lambda x.x
\end{aligned}
$$

Thus, `basic` supplies meanings to the metaterms of Peano arithmetic, as well as allowing us to use annotation in our metaterms analogous to the wave-front and wave-hole annotation of *CLAM* (see appendix A). In all examples from now on, we assume that the function environment and lifting tactic include at least these definitions.

Having thus defined `eval`, and obtained appropriate `env`'s and `f_env`'s, `e` and `f`, for the goal at hand, we derive the vertical arrows of fig. 3 1 by showing that $\mathtt{eval}(\text{“}s\text{”}, e, f) = s$ `in pnat` and $\mathtt{eval}(\text{“}t\text{”}, e, f) = t$ `in pnat`. These object level proof obligations are typically carried out by evaluating both sides to give an identity.

In order to derive the horizontal arrow of fig. 3 1, the *correctness goal*, we must show that the meta-operation, a member of the type `rewrite` defined by

```
rewrite:  complete
⊢ U1
Extract:  metaterm → metaterm
```

used to rewrite "s" to "t", preserves meaning. This has been done by defining the subset type

$$\begin{aligned}
&\texttt{simp: complete}\\
&\vdash \texttt{f_env} \rightarrow \texttt{U1}\\
&\texttt{Extract: } \lambda f.\{z : \texttt{rewrite} | \forall m : \texttt{metaterm}.\forall e : \texttt{env}.\\
&\qquad \texttt{eval}(m, e, f) = \texttt{eval}(z(m), e, f) \texttt{ in pnat}\}
\end{aligned}$$

parameterized over `f_env`'s, and showing that the meta-operation is also a member of $\texttt{simp}(f)$. Typically, a rewrite `R` will have a corresponding *correctness theorem* called `R_wff` stating $\vdash$ `R in simp(F)` for a given `f_env F`. This operation is summed up in the main lemma embodying the reflection mechanism:

$$\begin{aligned}
&\texttt{reflect: complete}\\
&\vdash \forall f : \texttt{f_env}.\forall s : \texttt{simp}(f).\forall t : \texttt{pnat}.\forall m : \texttt{metaterm}.\forall e : \texttt{env}.\\
&\quad t = \texttt{eval}(m, e, f) \texttt{ in pnat} \rightarrow t = \texttt{eval}(s(m), e, f) \texttt{ in pnat}
\end{aligned}$$

These three operations have been combined into an Oyster tactic, `reflect_tac/4`, taking as arguments the simplifier to employ, the `f_env` to use, the position of the subterm to lift, and an Oyster tactic which will be used to attempt to prove the correctness goal `ref_wfftac/0` is provided for the case mentioned above where there is a corresponding correctness theorem. Since we obtain t only by evaluating $\texttt{eval}(\text{“}t\text{”}, e, f)$, and since fully evaluating this typically unfolds definition instances we would like to retain, t is calculated from “t” by the routine `unlift_term/3` using the previously derived a-list.

In synthesis proofs we shall want to explicitly assert the correctness of a `basic` rewrite in all `env`'s. For this we define the shorthand relating the input term i to the output term o:

$$i \sim o \leftrightarrow \forall e : \texttt{env}.\texttt{eval}(i, e, \texttt{basic}) = \texttt{eval}(o, e, \texttt{basic}) \texttt{ in pnat}$$

Note that $\sim$ is a congruence relation.

It should be noted that the reflection mechanism is constructed completely within OYSTER's object theory — no new inference rules have been added and no "because" lemmas used — and is therefore a definitional extension of OYSTER, i.e.

no theorems can be proved using the mechanism which could not be proved previously. Problems may arise when the lifting and evaluation operations are not inverses (ideally we should parameterize both over a common representation of the theory to ensure a match), but whilst a mismatch may result in the inability to derive the equalities represented by the vertical arrows, it cannot result in a non-theorem.

In summary, using the reflection mechanism described above, we regard a tactic as a member of `rewrite`, which additionally is a member of `simp`(F) for a some `f_env` F. These are the two types of goal we shall need to prove in order to synthesize tactics.

We list the logical objects (definitions, theorems, etc.) which comprise the reflection mechanism described above in appendix D. Many of the latter theorems there appear only implicitly, e.g. those used by OYSTER tactics.

3.3.1 Concrete example

To illustrate the general case we now show how the reflection mechanism can be used to help prove a typical theorem in OYSTER. In this example we use the simplifier `term_of(assoc_simp)` which right associates the top-level `plus`'s in a metaterm, and for which we have the theorem:

```
assoc_simp_wff:  complete
⊢ term_of(assoc_simp) in simp(basic)
```

This allows us to use the `ref_wfftac` to solve the correctness goal as described above. Suppose that we arrive at the following point in a proof:

```
example:  [1,1,1] incomplete autotactic(idtac)
x : pnat, y : pnat, z : pnat
⊢ double(plus(plus(x, y), double(z))) = double(plus(x, plus(y, double(z)))) in pna
by _
```

We can see that the left- and right-hand sides are identical but for the association of the `plus`'s. We can therefore use our simplifier to rewrite the left-hand

side, resulting in an identity which is easily handled by the autotactic (which is switched off here). We apply `reflect_tac` with the arguments shown below. The object term is abstracted to the variable $v0$, and its meta-representation to $v1$. Hypotheses $v2$, $v4$ and $v3$ show respectively the original object term, its lifted representation (including the label $ref4$ for the subterm `double(`z`)` and the rewritten object term.

`example: [1,1,1] complete autotactic(idtac)`

x : `pnat`, y : `pnat`, z : `pnat`

⊢ `double(plus(plus(`x`,`y`),double(`z`))) = double(plus(`x`,plus(`y`,double(`z`)))) in pnat`

`by reflect_tac(term_of(assoc_simp),basic,[1,1,1],TRY ref_wfftac)`

`[1] complete`

$v0$: `pnat`,

$v2$: $v0$ = `plus(plus(`x`,`y`),double(`z`)) in pnat`,

$v1$: `metaterm`,

$v4$: $v1$ = `tbin('+',tbin('+',tvar('x'),tvar('y')),tvar('ref4')) in metaterm`,

$v3$: $v0$ = `plus(`x`,plus(`y`,double(`z`))) in pnat`

⊢ `double(plus(`x`,plus(`y`,double(`z`)))) = double(plus(`x`,plus(`y`,double(`z`)))) in pnat`

In (Howe, 1988a) it was noted that one of the main obstacles to everyday use of such a reflection mechanism was the time taken to continually reprove well-formedness subgoals. By abstracting the terms involved to variables, and by using the OYSTER rule

$h : a = b$ `in` $t \vdash a$ `in` t

`by hyp(h)`

which is not present in NuPRL, we need prove the well-formedness goals only once.

3.4 Extensions to the mechanism

The above constitutes a "bare-bones" reflection mechanism. It is restricted in real use in several ways: only unfailing rewrites are considered; it is not clear how we can to use rewrites outside their explicitly given `f_env`. In adding the following extensions to overcome these limitations we are guided by the intention of using *CLAM* in the resulting setting. Thus, to facilitate *CLAM*'s algebraic style of reasoning, the formalism should be as simple, transparent, natural and uniform as possible. To make *CLAM*'s task as simple as possible the emphasis is on "compiling out", as far as possible, object-level technicalities, to arrive at as simple an algebra of tactics as possible wherever something doesn't *have to* be done at synthesis time we have compiled out into a definition, lemma etc. The overall result of this is that the set of wave-rules *CLAM* must reason with satisfy the above criteria.

3.4.1 Failure

For the simple mechanism it was stated that tactics were members of the type `metaterm` → `metaterm`. Thus all tactics would "do" something, even though it might be returning a term unchanged. Usually, however, proof systems implementing tactics have a well-defined notion of tactic failure in the case of NuPRL, ML's exception mechanism is used; in OYSTER, the underlying Prolog's notion of success/failure is employed. Failure is an important means of passing information in its own right and a prerequisite if we are to build tactics in a modular fashion using tacticals.

We implement failure in the same manner as (Howe, 1988a). If we have a partial function, $f : a \rightarrow b$, we encode this in type theory as a function in $a \rightarrow ?b$ where $?$ is defined as the extract of

```
partial:  complete
          ⊢ U1 → U1
```

Extract: $\lambda t.(t|\texttt{atom})$

A success is now returned as a left injection of the result, a failure as a right injection of an error message in `atom`. We use the syntactic sugar `ok`(x), `failwith`('message') and `fail` below. Like Howe, we provide further sugared functions for manipulating "partial" functions and types (see appendix).

Using this notion of failure, a tactic is now a member of `metaterm` → `?metaterm`. We can lift tactics of the old type using the functional `total`.

3.4.2 Monotonicity and decidability

One of the biggest problems to overcome in synthesizing tactics formally is the localized nature of a synthesis theorem vs the global applicability of a tactic, i.e. a theorem has a theory as a context, whereas (most) tactics can be used in an arbitrary theory. This shows up in our case where we have a tactic in `simp(F)`. We should at least expect this tactic to be applicable to all extensions of `F`, i.e. tactics should be *monotonic*.

A large part of (Howe, 1988a) is devoted to showing the monotonicity of relevant notions, and "respect for environment extensions will be built into the definition of correctness of inference procedure." We take a more simple-minded, limited approach. Since we are mainly interested in tactics which manipulate wave annotation, and where other function symbols go mostly uninterpreted, most of our tactics will be in `simp(basic)`. When we come to apply such tactics in the context of a particular `f_env F`, we give as the tactic argument to `reflect_tac/4` (see above) `ref_wfftac_using/1`. This will expect to find a theorem stating the inclusion

$$\vdash x : \texttt{simp(F)} \rightarrow x \texttt{ in simp(basic)}$$

In practice this approach should not present a problem. One could even add invisibly a mechanism which extends `F` and updates such a theorem each time a definition is loaded/defined.

Besides the "globality" of the object logic in a proof system, we must also consider that of the tactic system itself NuPRL contains many global variables and *CLAM* has a large data base of wave-rules etc. Again, we choose a simple-minded solution. We encode a *CLAM*-like data base by providing decidability theorems for the relevant predicates. We distinguish decidability by giving it an explicit definition:

$$\texttt{decidable:}\quad \texttt{complete}$$

$$\vdash \texttt{U1} \rightarrow \texttt{U1}$$

$$\texttt{Extract:}\quad \lambda p.(p \vee (\neg p))$$

Now, to encode a predicate such as `is_defined_as`, we would expect a suitable theorem asserting

$$\vdash \forall f : \texttt{atom}.\delta(\exists m : \texttt{metaterm.is\_defined\_as}(f, m))$$

Functional predicates would have a more straight-forward form. Again, much as *CLAM*'s library mechanism updates its predicate data base, so the corresponding mimicking theorems can be updated. We call this *load-time simulation* and describe it in section 3.5 below.

3.4.3 An algebra of tactics

In (Paulson, 1983a), Paulson notes that "conversions may be amenable to algebraic reasoning." It is an algebra of rewrite tactics which we must ultimately present to *CLAM* and use in synthesis proofs. At the bottom level we will specify primitive tactics using the definitions which naturally arise, such as those of the *CLAM* method language. As we build larger tactics we expect to make use of those tactics previously defined, putting them together in the modular LCF style of Paulson. And rather than give them primitive specifications, we expect to be able to determine their behaviour from the specifications of their subparts and of the gluing *tacticals*.

We follow Howe's encoding of Paulson-style tacticals for use with the reflection mechanism. We have:

`f then g`: apply f, then apply g; fail if either does.

`f orelse g`: if f succeeds apply f; otherwise apply g.

`try f`: if f succeeds apply f; otherwise do nothing.

`sub f`: apply f to immediate subterms, failing if any application does.

`idtac`: succeed by doing nothing.

We can give simple (2nd order) equations describing the behaviour of such tacticals, for example:

$$f(x) = \texttt{ok}(y) in ?(\texttt{metaterm}) \rightarrow (f \texttt{ then } g)(x) = g(y) \texttt{ in } ?(\texttt{metaterm})$$

$$f(x) = \texttt{failwith}(y) in ?(\texttt{metaterm}) \rightarrow (f \texttt{ then } g)(x) = \texttt{failwith}(y) \texttt{ in } ?(\texttt{metaterm})$$

By using the wave-rules for these tacticals and a well-chosen set of inductions schemes we can use *CLAM* to reason about the behaviour of compound tactics built in this way.

3.5 Load-time simulation

One problem that is common to any reflection mechanism used in a theorem proving environment is the need to take account of context. One rarely proves a theorem in isolation. More usually there is a large body of definitions and previously proved theorems present, and possibly meta-theoretic information, all of which influence the proving process. This context-setting is typically implemented by a library mechanism. The reflection mechanism, based on the logic used rather than the theorem prover, is unable to represent the library mechanism. Various ways of overcoming this problem and one direct attack (??) are described in chapter 6.

This problem is compounded in our case since our tactics, for use with *CLAM*, must refer to an extensive data base of meta-level information — wave rules, induction schemas, conditional rewrites etc. We have chosen to partially *simulate* the *CLAM* data base. Our method is more tractable and far less complex than

that described in section ??, but at the cost of a certain logical inadequacy. We first describe our approach in detail, then its inadequacy and how it affects us in practice.

When a definition, theorem etc., is loaded into *CLAM* the action has two consequences. One or several object-level definitions and/or theorems are loaded and become accessible in the OYSTER logic. Simultaneously *CLAM*'s meta-level data base is updated. The effect is similar no matter what kind of object is loaded, so we shall described only how definitions are treated for simplicity.

A detailed description of Howe's work and a comparison with ours can be found in section 6.3.

3.6 Discussion

Q: How can our tactics fail?
A: When their preconditions fail. We move all the conditions necessary to decide whether the tactic will fail or succeed into the precondition.

Q: Can we always do this?
A: No. We couldn't do it for, say, a theorem prover for FOPL since it is undecidable. However, we are considering only terminating tactics and we can if necessary move the whole tactic into the preconditions. E.g.

$$\forall i.(\exists o2.\mathtt{effects}(i, o2)) \rightarrow \exists o.\mathtt{effects}(i, o)$$

Q: What do we have to prove to class an (atomic) tactic as synthesized?
A: We have to prove the theorem:

$$\vdash \forall i.\mathtt{preconds}(i) \rightarrow \exists o.\mathtt{effects}(i, o) \wedge i \sim o \tag{3.1}$$

Almost as important is the decidability of the preconditions:

$$\vdash \forall i.\delta(\mathtt{preconds}(i)) \tag{3.2}$$

Note that (3.2) allows us to transform (3.1) into

$$\vdash \forall i.\exists o.\texttt{preconds}(i) \rightarrow \texttt{effects}(i,o) \wedge i \sim o \tag{3.3}$$

Q: What advantages does (3.3) have over (3.1)?
A: First, it is in prenex form and more suited to *CLAM* (at least without context sensitive rewriting). Second, compare the extracts:

$$\lambda i.\lambda hyp. < o, \_ >$$

and

$$\lambda i. < o, \lambda hyp.\_ >$$

We see that it is much easier to get hold of o in the second; in fact we don't need to show the preconditions first! (Of course, o will not necessarily be a valid substitute for i if the preconditions do not hold. But we see below that this is still useful in the case of compiled tactics where we are interested only in the value of o.)

Q: So, if we have proved (3.1), and possibly (3.2), how do we get the actual tactic?
A: We use the reflection mechanism. We assume, as do our methods, that the correctness goal, $i \sim o$, always occurs as the topmost, right-hand conjunct of the succeedent in a synthesis goal (see (3.1) and (3)). With this we can show that the rewrite $i :\Rightarrow o$ is a valid rewrite for that environment. To apply it, i.e. substitute o for i in some goal all that is required is that we prove

$$\vdash \texttt{preconds}(i)$$

and use the primitive substitution rule. [There are OYSTER tactics to do all this transparently.] If we have proved (2) then we simply evaluate its extract: if it is a left injection then we do as above; if right we fail. We never try to use or evaluate the effects if the preconds(i) is not provable.

Q: Isn't it inefficient to replicate the effects in the preconditions?
A: No. Remember that the effects are purely declarative and, unlike *CLAM*, will never be "executed". We use them purely for their logical content.

Q: What about compiling these tactics to Prolog?

A: We can quite easily, given (2), compile a guaranteed-to-terminate (modulo floundering) Prolog program that is correct, i.e. succeeds exactly when `preconds(i)` is provable. We are also able to easily calculate the unlifted version of `o`. Since this is an informal "running" of the preconditions we are unable to directly infer the correctness goal.

This presents us with a paradigm for *abstract proof planners*. We can compile precondition decidability and output term into a *CLAM* method. The tactic supporting it results from the ordinary reflective case above. A successful plan is guaranteed to succeed and may be found considerably quicker than evaluating the preconditions at the object level. Of course, to apply the tactic we will have to evaluate preconditions at the object level (to get at the correctness goals), *but only the ones picked out by the planner.*

Q: Is the correspondence between `preconds(i)` and `effects(i,o)` and *CLAM*'s precondition and effects slots precise?

A: No. `preconds(i)` is sufficient to decide whether a tactic applies (unlike *CLAM*). `effects(i,o)` are purely declarative since the output, `o`, is calculated in evaluating the extract. We use `effects(i,o)` to chain proofs together (in synthesizing compound tactics) rather than rely on the explicit structure of `o` as *CLAM* and the abstract planner above do. [In fact, *CLAM*'s slots would better be termed *analysis* (taking the input apart) and *synthesis* (no relation: putting the output together). It was intended originally to have a more declarative version of the effects slot but tractability problems prevented this. Meta-level annotation (waves) were put explicitly in the input and outputs instead.]

Q: How do we build compound tactics?

A: We take a very simple approach at present, relying on a few properties of given tacticals. For example, given that `tac1` and `tac2` have, respectively, the following synthesis theorems:

$$\forall i.\exists o.\texttt{preconds1}(i) \rightarrow \texttt{effects1}(i,o) \wedge i \sim o$$
$$\forall i.\exists o.\texttt{preconds2}(i) \rightarrow \texttt{effects2}(i,o) \wedge i \sim o$$

we automatically deduce for respectively `tac1 ORELSE tac2`, `tac1 THEN tac2` and `REPEAT tac1` the following:

$$\forall i.\exists o.(\texttt{preconds1}(i) \vee \texttt{preconds2}(i)) \rightarrow (\texttt{effects1}(i,o) \vee \texttt{effects1}(i,o)) \wedge i \sim o$$

$$\forall i.\exists o.(\texttt{preconds1}(i) \wedge (\exists t.\texttt{effects2}(i,t) \rightarrow \texttt{preconds2}(l))) \rightarrow$$
$$\exists t.\texttt{effects1}(i,t) \wedge \texttt{effects2}(t,o) \wedge i \sim o$$

$$\forall i.\exists o.\texttt{true} \rightarrow (\neg\texttt{preconds1}(o)) \wedge i \sim o$$

The last property is not valid in type theory since `REPEAT tac1` may not terminate. We "because" this lemma and implement repeat by iterating `tac1` a large (but finite) number of times.

Q: How does non-determinism arise?

A: The main way in which non-determinism arises is via the `ORELSE` tactical. We suppose that all *CLAM* predicates are used in such a manner that they are completely deterministic. If we wish to parameterize non-determinism then, as in ML tactics, we may iterate `ORELSE` over a list of alternatives.

Q: How is the environment used?

A: This is used to store information which is likely to change in the course of a proof. Unfortunately, it assumes a fixed theory throughout a session. This isn't as bad as it seems since we can do the necessary well-typedness proofs once and for all at load time.

Chapter 4

Tactic synthesis

4.1 Introduction

In this chapter we develop proof plans for tactic synthesis, that is to prove theorems which assert the existence of a correct rewrite in the formalism developed in the previous chapter. One should ask what is reasonable to expect of such a programme. It is unlikely to be as successful as a human at writing tactics since it possesses none of the high-level semantic guidance of how a tactic is to be used and how it is integrated into a library of existing and future tactics. It would also be optimistic to expect large inference steps from highly declarative specifications to executable tactics for the same reason, and also from looking at the state of the art in the simpler work of verifying recursive programs (Bundy *et al*, 1991; Boyer & Moore, 1979). What we can expect is a useful array of techniques which, under human guidance, can formally construct provably correct rewriting tactics satisfying specifications written in a declarative, high-level language.

We have split tactic synthesis into two parts: the automatic synthesis of atomic tactics and the partial automation of the verification of compound tactics from already-synthesized tactics. We have found, after synthesizing several atomic tactics, a number of principled extensions to the existing $CLAM$ method set were required. Below we motivate the introduction of these with a detailed example

showing why and where they are needed. We have tried, as far as possible, to make all extensions monotonic, i.e. extending *CLAM*'s proving ability to include tactic synthesis rather than redirecting it. The principle extensions we make are the use of higher-order (h.o.) wave rules and context-sensitive rewriting. We also list more minor ways in which we have extended methods.

In our representation of tactics and the predicates involved their specifications we have attempted to hide all the low-level features (such as association lists, matching, non-determinism) from the synthesis theorems. This has been achieved by using assuming that the user has already provided lemmas stating the necessary high-level properties of these predicates (such as correctness and decidability in various modes). By putting as much as possible of the computational content of a predicate into theorems of its high-level behaviour, less of this occurs in the synthesis proof itself, so leading to more efficient extracted tactics.

4.2 Synthesis proofs

We have shown in chapters 2 and 3 the well known methods by which programs in general and tactics in particular may be specified in a constructive type theory, and how, once these specification theorems have been proved, programs may be extracted. In this section we explain in detail how we derive such specifications directly from *CLAM* method specifications. We also state what it means to have synthesized a tactic and some properties of the theorems.

We consider here *CLAM* methods which have been simplified in keeping with our restriction to rewrite tactics over `metaterms`; in particular, the input and output slots are now occupied by subterms of a sequent in place of the sequent itself. Recall that a method specification has the form:

```
method(name(Args),          % name and arguments
       I,                   % input slot
       O,                   % output slot
```

```
Preconds(I,X,Y),          % preconditions
Effects(I,O,X,Z),         % effects
?                         % tactic we wish to synthesize
).
```

We have made explicit the Prolog variables present in this specification. In a formal specification these approximate to prenexable existential variables. Note also that we have not disallowed the sharing of variables between preconditions and effects. `X`, `Y` and `Z` denote respectively variables common to preconditions and effects, private to preconditions, and private to effects. `Args` may contain any or all of these variables.

We require the following to hold:

1. The preconditions should not mention the output `O` and should be sufficient to decide whether the tactic is applicable, i.e. if the preconditions are true then we can find `O` such that the effects are satisfied.

2. The preconditions should, if possible, be decidable. This property is used in transforming proofs so that extracts become "cleaner", in compiling pseudo tactics to Prolog, and in synthesizing compound tactics. (See chapter 5.)

3. The metaterms constituting the input and output have the same meaning, i.e. the OYSTER terms they represent are provably equal. This is our correctness goal, $i \sim o$, from chapter 3. It ensures that the tactic is correct at the object level.

4. We do not require that the effects uniquely determine the output.

We can now formalize criteria (1) and (2) by the assertions:

$$\forall i : \texttt{metaterm}.\forall x, y.\texttt{preconds}(i, x, y) \rightarrow$$
$$\exists o : \texttt{metaterm}.\exists z.\texttt{effects}(i, o, x, z) \wedge i \sim o \quad (1)$$
$$\forall i : \texttt{metaterm}.\forall args.\delta(\exists x, y - args.\texttt{preconds}(i, x, y)) \quad (2)$$

where x, y and z are parameters of fixed type dependent on the theorem, and $x, y - args$ denotes variables occurring in x or y and not in $args$. [Note: like X, Y and Z, we use x, y and z as a shorthand to indicate tuples of variables.] We call the first the *synthesis theorem* and the second the *decidability theorem*. The

reason for the form of the latter is that we must search for the free variables of the preconditions which are not input explicitly as arguments. The decidability assertion states that this search terminates with success or failure. We will see in chapter 5 that this search can be moved to the execution of a Prolog program. For this reason we will always be interested in proving a second form of decidability theorem where $args$ is x, y, i.e.

$$\forall i : \texttt{metaterm}.\forall x, y.\delta(\texttt{preconds}(i, x, y))$$

Once we have proved these theorems it is simple to construct a tactic. Given an input i and arguments $args$, we evaluate the result of applying the extract of the decidability theorem to these. If it is a right injection the tactic fails. If it is a left injection we may obtain the witnesses for $x, y - args$ by taking projections and apply the extract of the synthesis proof to these and $args$. Evaluating this term gives us the output o which we know will be correct, i.e. it satisfies the `effects` and $i \sim o$. If the decidability theorem for a tactic is unavailable then the proof obligation $\exists x, y - args.\texttt{preconds}(i, x, y)$ remains as a subgoal. This process can be made transparent using OYSTER tactics. For a tactic called `tac` we follow the convention of calling the synthesis theorem `tac_synth` and the decidability theorem `tac_dec`. The second form of decidability theorem is called `tac_comp`.

`tac_comp` is useful in several ways. We will show later in section 4.5 how it is used for putting synthesized tactics together to form larger ones. It also allows us to make the output o independent of the way in which the preconditions are proved. For, given `tac_comp`, we can show the equivalence of the synthesis theorem with:

$$\forall i : \texttt{metaterm}.\forall x, y.\exists o : \texttt{metaterm}.\exists z.\texttt{preconds}(i, x, y) \rightarrow$$
$$\texttt{effects}(i, o, x, z) \wedge i \sim o$$

The extract of this proof

$$\lambda i.\lambda x, y. < o, \_ >$$

is much "cleaner" than that of the synthesis theorem we may extract o simply by taking the first projection after applying to the arguments. We have not considered this aspect of efficiency closely. Work has been done (Paulin-Mohring, 1989;

Constable *et al*, 1986) on the elimination of computationally uninteresting terms from a proof extract. We believe that the use of OYSTER's subset types could be applied to the work in this thesis but have not had time to verify this.

`tac_comp` will be used in chapter 5 to compile a pseudo tactic `prog` to simulate `preconds`. A correctness theorem there will show that:

$$\vdash \texttt{preconds}(i, x, y) \iff \vdash_{\text{SLD}} \texttt{prog}(I, X, Y, O)$$

We have then come full circle since `prog` can be used as a pseudo-tactic, as are *CLAM* methods, allowing the efficient planning of proofs. The difference now is that we *know* we are able to reify the planned proof with tactics.

For each pre-defined predicate of interest in *CLAM*'s methodical language we either provide a direct analogue in type theory or, in cases where this is not possible, e.g. routines for manipulating quantifiers which our formalism does not permit, functions carrying out equivalent operations, e.g. `exp_at` and `replace` in the example which follows. For each of these we will have a set of lemmas: for predicates we will have a lemma asserting decidability in various modes; for functions and predicates we will have lemmas asserting correctness where suitable. There will also be the usual lemmas relating functions and predicates to each other.

4.3 Example

In this section we go through an example synthesis proof as if *CLAM* were attempting to build a plan for it. At certain points we highlight shortcomings of the present *CLAM* system and note how one might fix these. Later we shall present a rational reconstruction of these fixes and see that, in essence, they represent extensions to the present set of *CLAM* methods rather than a wholesale introduction of new ones. This is in keeping with our hopes of monotonically extending general induction theorem proving techniques to tactic synthesis.

The examples we present are the synthesis and decidability theorems for the `eval_def` tactic. This is used in its full form by *CLAM* to rewrite a definition

to its definiens. To keep the example clear, and because we are anyway working with a rewrite version of this tactic, we have simplified `eval_def`'s definition. We now have:

```
method(eval_def(Pos,Exp),
       I,
       O,
       [exp_at(I,Pos,Exp), def\_eqn(Exp,NewExp)],
       [replace(I,Pos,NewExp,O)],
       ?
      )
```

The definitions for `exp_at` and `replace` and the theorems we assume about the simulated predicate `def_eqn` can be found in appendix K. We do not show the full proof below since much of it is repetitive. In particular we give only one base and one step case of the main induction. The base cases are almost identical. In the first example we show only the second step case: this involves multi-hole rippling (see section A.3), the other step case going through with similar caveats.

As explained in the previous section, the two theorems which we require for the synthesis of a tactic to satisfy these specifications are:

$$\vdash \forall i, x, n : \texttt{metaterm}, pos : \texttt{posn}.(\texttt{exp\_at}(i, pos, x) \wedge \texttt{def\_eqn}(x, n)) \rightarrow$$
$$\exists o : \texttt{metaterm}.\texttt{replace}(i, pos, n, o) \wedge i \sim o$$
$$\vdash \forall i : \texttt{metaterm}.\delta(\exists x, n : \texttt{metaterm}, pos : \texttt{posn}.\texttt{exp\_at}(i, pos, x) \wedge \texttt{def\_eqn}(x, n))$$

First induction: $i : \texttt{metaterm}$

Recursion analysis, after examination of the definitional wave rules for `exp_at` and `replace` (see glossary, appendix K) suggests `metaterm` induction on i as the least flawed.

Base case: $\texttt{tvar}(v)$

Let us look at the first base case:

$$v : \texttt{atom} \vdash \forall x, n : \texttt{metaterm}, pos : \texttt{posn}.(\texttt{exp\_at}(\texttt{tvar}(v), pos, x) \wedge \texttt{def\_eqn}(x, n)) \rightarrow$$
$$\exists o : \texttt{metaterm}.\texttt{replace}(\texttt{tvar}(v), pos, n, o) \wedge \texttt{tvar}(v) \sim o$$

We can now rewrite both `exp_at` and `replace` according to their definitions:

$$\ldots \vdash \forall x, n : \texttt{metaterm}, pos : \texttt{posn}.$$
$$(pos = \texttt{nil in posn} \wedge x = \texttt{tvar}(v) \texttt{ in metaterm} \wedge \texttt{def\_eqn}(x, n)) \rightarrow$$
$$\exists o : \texttt{metaterm}.(pos = \texttt{nil in posn} \wedge n = o \texttt{ in metaterm}) \wedge \texttt{tvar}(v) \sim o$$

As (Boyer & Moore, 1979) point out, equalities should be eliminated whenever possible - they usually indicate a substitution which we can carry out and forget about. It also obviously simplifies a formula. More importantly, as we shall see in the step case below, they may form part of a wave front and their elimination is an essential part of unblocking. If the equality is between a variable and a term free in that variable, we can often eliminate the quantifier binding that variable as well. This depends on the position of the equality in the formula in a way we shall make clear later. In this case the positions of the equalities involving both *pos* and x are such that we can eliminate their quantifiers. Thus, our much-simplified goal is now:

$$\ldots \vdash \forall n : \texttt{metaterm.def\_eqn}(\texttt{tvar}(v), n) \rightarrow$$
$$\exists o : \texttt{metaterm}.(\texttt{nil} = \texttt{nil} in \texttt{posn} \wedge n = o \texttt{ in metaterm}) \wedge \texttt{tvar}(v) \sim o$$

We can simplify the equality $\texttt{nil} = \texttt{nil}$ and eliminate the the quantifier $\exists o$ since this satisfies too our position criterion. This step, since it involves instantiating the existential variable o with the value n, is our first explicit point of synthesis. The result is:

$$\ldots \vdash \forall n : \texttt{metaterm.def\_eqn}(\texttt{tvar}(v), n) \rightarrow \texttt{tvar}(v) \sim n$$

At this point we note that the goal matches the correctness statement for the simulated predicate `def_eqn`, and are finished.

Step case: $\boxed{\texttt{tbin}(f, \underline{a_1}, \underline{b_2})}$

We now examine the second step case, that involving $\boxed{\texttt{tbin}(f, \underline{a_1}, \underline{b_2})}$. Our goal is:

$$
\begin{array}{l}
f : \texttt{atom}, a : \texttt{metaterm}, b : \texttt{metaterm}, \\
h1 : \forall x, n : \texttt{metaterm}, pos' : \texttt{posn}.(\texttt{exp\_at}(a, pos, x) \wedge \texttt{def\_eqn}(x, n)) \\
\qquad \rightarrow \exists o : \texttt{metaterm.replace}(a, pos, n, o) \wedge a \sim o \\
h2 : \forall x, n : \texttt{metaterm}, pos : \texttt{posn}.(\texttt{exp\_at}(b, pos, x) \wedge \texttt{def\_eqn}(x, n)) \\
\qquad \rightarrow \exists o : \texttt{metaterm.replace}(b, pos, n, o) \wedge b \sim o \\
\vdash \forall x, n : \texttt{metaterm}, pos : \texttt{posn}.(\texttt{exp\_at}(\boxed{\texttt{tbin}(f, \underline{a_1}, \underline{b_2})}, \lfloor pos \rfloor, x) \wedge \texttt{def\_eqn}(x, n)) \rightarrow \\
\qquad \exists o : \texttt{metaterm.replace}(\boxed{\texttt{tbin}(f, \underline{a_1}, \underline{b_2})}, \lfloor pos \rfloor, n, o) \wedge \boxed{\texttt{tbin}(f, \underline{a_1}, \underline{b_2})} \sim o
\end{array}
$$

Seconds induction (nested): $pos : \texttt{posn}$

Recursion analysis, again from the definitional wave rules and taking note of the position of sinks, suggests a nested induction on pos.

Base case: `nil`

In the base case we have, after expanding base case definitions:

$$
\begin{array}{l}
\ldots \vdash \forall x, n : \texttt{metaterm}.(x = \texttt{tbin}(f, a, b)\texttt{tbin in metaterm} \wedge def\_(x, n)) \rightarrow \\
\exists o : \texttt{metaterm}.n = o \texttt{ in metaterm} \wedge \texttt{tbin}(f, a, b) \sim o
\end{array}
$$

As in the previous base case, the equalities for x and o are in such positions that we can eliminate their quantifiers, instantiating o with n:

$$\ldots \vdash \forall n : \texttt{metaterm.def\_eqn}(\texttt{tbin}(f, a, b), n) \rightarrow \texttt{tbin}(f, a, b) \sim n$$

As in the base case we note that this is another instance of the correctness theorem for def_eqn.

Step case: $\boxed{h :: \underline{t_3}}$

We now move on to the step case of the pos induction. We have a new hypothesis, $h3$, in addition to the two already present from the induction on i:

$$
\begin{array}{l}
h : \texttt{pnat}, t : \texttt{pnat list}, \\
h3 : \forall x, n : \texttt{metaterm}.(\texttt{exp\_at}(\boxed{\texttt{tbin}(f, \underline{a_1}, \underline{b_2})}, t, x) \wedge \texttt{def\_eqn}(x, n)) \rightarrow \\
\qquad \exists o : \texttt{metaterm.replace}(\boxed{\texttt{tbin}(f, \underline{a_1}, \underline{b_2})}, t, n, o) \wedge \boxed{\texttt{tbin}(f, \underline{a_1}, \underline{b_2})} \sim o \\
\vdash \forall x, n : \texttt{metaterm}.(\texttt{exp\_at}(\boxed{\texttt{tbin}(f, \underline{a_1}, \underline{b_2})}, \boxed{h :: \underline{t_3}}, x) \wedge \texttt{def\_eqn}(x, n)) \rightarrow \\
\qquad \exists o : \texttt{metaterm.replace}(\boxed{(f, \underline{a_1}, \underline{b_2})}, \boxed{h :: \underline{t_3}}, n, o) \wedge \boxed{\texttt{tbin}(f, \underline{a_1}, \underline{b_2})} \sim o
\end{array}
$$

We can now ripple out exp_at in this goal. Note that we can only ripple the different fronts (1 and 3/1 and 2) since in each case the wave fronts for 3 are inside a sink for 1 or 2. Since the wave front for 3 is moved inside a wave hole for 1/2 it is erased throughout the goal since it is no longer possible to fertilize it, i.e. we have no hope of using h3:

$$
\begin{array}{l}
\ldots \vdash \forall x, n : \texttt{metaterm}. \\
(\boxed{((h = 0 inpnat \wedge \underline{\texttt{exp\_at}(a, t, x)}_1) \vee (h = 1 inpnat \wedge \underline{\texttt{exp\_at}(b, t, x)}_2))} \\
\qquad \wedge \texttt{def\_eqn}(x, n)) \rightarrow \ldots
\end{array}
$$

At this point the rippling is blocked. To continue we require the presence of propositional wave rules, viz.

$$
\begin{array}{l}
\boxed{(\underline{A}_1 \vee \underline{B}_2)} \wedge C \mapsto \boxed{\underline{(A \wedge C)}_1 \vee \underline{(B \wedge C)}_2} \\
\boxed{(\underline{A}_1 \vee \underline{B}_2)} \rightarrow C \mapsto \boxed{\underline{(A \rightarrow C)}_1 \wedge \underline{(B \rightarrow C)}_2}
\end{array}
$$

Of more interest is the fact that we also need a ripple corresponding to the wave rule:

$$
\forall x : T.\boxed{\underline{A}_1 \wedge \underline{B}_2} \mapsto \boxed{\underline{\forall x : T.A}_1 \wedge \underline{\forall x : T.B}_2}
$$

This is a higher-order wave rule where we need to take account of such things as variable capture and dependence and perform the necessary α-conversion. Assuming that we have all these rules we may now ripple the goal to:

$$
\ldots \vdash \underline{\underline{\boxed{(\forall x, n : \texttt{metaterm}.(\boxed{(h = 0 inpnat \wedge \underline{\texttt{exp\_at}(a, t, x)}_1)}}}}
$$
$$
\underline{\underline{\qquad \wedge \texttt{def\_eqn}(x, n)) \rightarrow \exists o : \texttt{metaterm}.\ldots)_1 \wedge \underline{\ldots}_2}}
$$

Again, wave fronts inside wave holes are erased if they do not have identical indices. Note that a convenient, but not necessary, operation at this point would be to split the goal by performing an $\wedge$-introduction. This behaves as a sequent-level wave cancellation rule. [We can do likewise for the $\vee$ connective, trying to prove one of the subgoals.] Let us assume we have performed such an operation for reasons of clarity. Thus our goal is:

$$\ldots \vdash \forall x, n : \texttt{metaterm}.(\boxed{(h = 0 inpnat \wedge \underline{\texttt{exp\_at}(a, t, x)}_1)} \wedge \texttt{def\_eqn}(x, n)) \rightarrow$$
$$\exists o : \texttt{metaterm.replace}(\boxed{\texttt{tbin}(f, \underline{a}_1, b)}, h :: t, n, o) \wedge \boxed{\texttt{tbin}(f, \underline{a}_1, b)} \sim o$$

We can now eliminate the equality on h. This time we do not eliminate any quantifier but we note which occurrences in the goal can be soundly rewritten using this equality. This is a technique we have called context-sensitive rewriting and explore in detail in section 4.7. We can eliminate the equality itself on similar grounds to the quantifier elimination case. This time, however, we are in a step case. It would not be useful to eliminate equalities if we disrupt the skeleton, so we add an extra proviso to prevent this. In this case the equality is in a wave front and the operation can be seen as an extension to the unblocking method. The resulting goal is now:

$$\ldots \vdash \forall x, n : \texttt{metaterm}.(\texttt{exp\_at}(a, t, x) \wedge \texttt{def\_eqn}(x, n)) \rightarrow$$
$$\exists o : \texttt{metaterm.replace}(\boxed{\texttt{tbin}(f, \underline{a}_1, b)}, 0 :: t, n, o) \wedge \boxed{\texttt{tbin}(f, \underline{a}_1, b)} \sim o$$

The only ripple we can perform at this point is an existential one using the wave rule for replace. This instantiates o to tbin(g,c,d), introducing in the process the new existential variables g, c and d:

$$\ldots \vdash \ldots \rightarrow \exists g : \texttt{atom}.\exists c, d : \texttt{metaterm}.$$
$$(f = gin\texttt{atom} \wedge ((0 = 0 inpnat \wedge b = dinmetaterm \wedge \texttt{replace}(a, t, n, c))$$
$$\vee (0 = s(0) inpnat \wedge a = cinmetaterm \wedge \texttt{replace}(b, t, n, d))))$$
$$\wedge \boxed{\texttt{tbin}(f, \underline{a}_1, b)} \sim \boxed{\texttt{tbin}(g, \underline{c}_1, d)}$$

After boolean simplification the result is:

$$\ldots \vdash \ldots \rightarrow \exists g : \texttt{atom}.\exists c, d : \texttt{metaterm}.$$
$$(f = gin\texttt{atom} \wedge (b = dinmetaterm \wedge \texttt{replace}(a, t, n, c)))$$
$$\wedge \boxed{\texttt{tbin}(f, \underline{a}_1, b)} \sim \boxed{\texttt{tbin}(g, \underline{c}_1, d)}$$

We can now ripple out the $\sim$ goal. We can also eliminate the the equalities on g and d and their quantifiers, instantiating them respectively with the values f and b. If we assume the presence of the reflexivity theorem for $\sim$ the result is:

$$\ldots \vdash \ldots \rightarrow \exists c : \texttt{metaterm.replace}(a, t, n, c) \wedge a \sim c$$

This can now be fertilized with hypothesis $h1$. The proof is finished. An elided form of the extract can be found in appendix H.

We now go on to finish off the synthesis process by similarly annotating the proof of the decidability of the preconditions of `eval_def`. Since the `tbin` step case does not illustrate anything that has not already been raised above with respect to multi-hole rippling, we will use the simpler `tuna` step case in this example. Our starting goal is:

$$\vdash \forall i : \texttt{metaterm}.\delta(\exists x, n : \texttt{metaterm}, pos : .\texttt{exp\_at}(i, pos, x) \wedge \texttt{def\_eqn}(x, n))$$

The only induction suggested by recursion analysis here is on i, arising from `exp_at`. The first base case simplifies to:

$$v : \texttt{atom} \vdash \delta(\exists x, n : \texttt{metaterm}, pos : \texttt{posn}.$$
$$(pos = \texttt{nil in posn} \wedge x = \texttt{tvar}(v) \texttt{ in metaterm}) \wedge \texttt{def\_eqn}(x, n))$$

Since they satisfy our position criterion we can now eliminate the existential quantifiers for pos and x with respectively `nil` and `tvar`(v). It is worth noting at this point that a rewrite inside $\delta(P)$ must be two-way since it is short for $P \vee \neg P$, i.e. each subterm occurs with both polarities. Since our quantifier elimination is equivalence-preserving, this is permissible here. After the elimination our base case looks like:

$$\ldots \vdash \delta(\exists n : \texttt{metaterm.def\_eqn}(\texttt{tvar}(v), n))$$

We expect this lemma to be present already for the simulated predicate `def_eqn` for it asserts its termination when used in the mode (+,-).

In the step case we have the goal:

$f : \texttt{atom}, a : \texttt{metaterm},$

$h1 : \delta(\exists x, n : \texttt{metaterm}, pos : \texttt{posn.exp\_at}(a, pos, x) \wedge \texttt{def\_eqn}(x, n))$

$\vdash \delta(\exists x, n : \texttt{metaterm}, pos : \texttt{posn.exp\_at}(\boxed{\texttt{tuna}(f, \underline{a_1})}, pos, x) \wedge \texttt{def\_eqn}(x, n))$

The only thing suggested by this goal is an existential ripple on pos. Since we require this to be a two-way rewrite we must use information from the schema corresponding to the ripple to arrive at:

$$\ldots \vdash \delta(\exists x, n : \texttt{metaterm}.\boxed{(\texttt{exp\_at}(\texttt{tuna}(f,a),\texttt{nil},x) \wedge \texttt{def\_eqn}(x,n))} \\ \boxed{\vee \exists h : \texttt{pnat}, t : \texttt{posn}.(\texttt{exp\_at}(\boxed{\texttt{tuna}(f,\underline{a}_1)},\boxed{h :: \underline{t}1},x) \wedge \texttt{def\_eqn}(x,n))_1})$$

Now, applying definitions, eliminating h with witness 0, rippling and simplifying we get:

$$\ldots \vdash \delta(\exists x, n : \texttt{metaterm}. \\ \boxed{\texttt{def\_eqn}(\texttt{tuna}(f,a),n) \vee \exists t : \texttt{posn}.(\texttt{exp\_at}(a,t,x) \wedge \texttt{def\_eqn}(x,n))_1})$$

Now, applying higher-order wave rules of the form

$$\exists x : T.\boxed{A \vee \underline{B(x)}} :\Rightarrow \boxed{A \vee \underline{\exists x : T.B(x)}}$$
$$\exists x : T.\boxed{A(x) \vee \underline{B(x)}} :\Rightarrow \boxed{\underline{\exists x : T.A(x)} \vee \underline{\exists x : T.B(x)}}$$

and using the rules for distributing δ over the propositional connectives we arrive at the fertilizable goal:

$$\ldots \vdash \boxed{\delta(\exists n : \texttt{metaterm}.\texttt{def\_eqn}(\texttt{tuna}(f,a),n)) \\ \wedge \underline{\delta(\exists x, n : \texttt{metaterm}, t : \texttt{posn}.\texttt{exp\_at}(a,t,x) \wedge \texttt{def\_eqn}(x,n))_1}}$$

The first conjunct is handled as above, by supposing the presence of an decidability lemma for def_eqn.

We now recap on the extensions required to *CLAM*. These were equality and quantifier elimination, context-sensitive rewriting and the use of higher-order wave rules. We also needed extensions to recursion analysis to allow for nested inductions involving sinks, and to existential rippling when two-way rewrites are necessary. This completes our two examples. In the next section we analyse these extensions were and attempt a rational reconstruction from these.

4.4 Synthesis of atomic tactics

Having illustrated our case with an example we can now attempt a rational reconstruction. We should note that such a generalization is justified since, although simple, `eval_def` has the characteristics of most *CLAM* tactics. The preconditions involve non-deterministically dismantling a goal whilst the effects use this information to build the output. We should also note that `exp_at` and `replace` have very similar recursion schemes. Moreover the nested `metaterm/posn` induction behaves like depth induction on `metaterm`.

The general purpose proof plan is well-suited to this form of proof with the provisos noted. In particular we have the problems of requiring higher-order ripples, context-sensitive rewriting and quantifier elimination. These are all detailed in there own sections below.

We also note some minor extensions required as follows:

1. Since we are working inside decidability statements we require all rewrites to be equivalences or equalities. This is not the case with present existential rippling. For example the ripple

$$\exists l : a\ \texttt{list}.P(\boxed{\underline{l}}) :\Rightarrow \exists h : a, t : a\ \texttt{list}.P(\boxed{h :: \underline{l}})$$

is not an equivalence. We require that the full equivalence be used:

$$\exists l : a\ \texttt{list}.P(\boxed{\underline{l}}) :\Rightarrow \boxed{P(\texttt{nil}) \vee \underline{\exists h : a, t : a\ \texttt{list}.P(\boxed{h :: \underline{l}})}}$$

Such case-splits can be automatically derived using the same schema information as is used by recursion analysis to chose inductions.

2. The strong fertilization submethod has been upgraded. It now takes into account an arbitrary quantifier prefix. We did not need such generality for our essentially $\forall\exists$ theorem, but a simple means of skolemizing, matching and deskolemizing was found to be easier to express for the general case. This submethod is given in the appendix G.

3. We have added a new method to handle, as a last resort, cases of a pure correctness, i.e. a goal where the only predicate is $\sim$. This is reflected down and *CLAM* given the object-level goal to attempt.

4. We have updated recursion analysis to take account of nested induction where multi-hole rippling can take place only because of a sink. This is illustrated by the `tbin` ripple on `replace` in the example above.

5. We have also updated the erasure of fronts which, by virtue of falling into differently indexed wave-holes are no longer feasible fertilization candidates. This reduces the search space.

Our formulation of wave rules is slightly unorthodox. Instead of writing several conditional one-way wave rules, we prefer to write unconditional, two-way rules containing equalities and propositional connectives. This would not normally be an advantage, but since we are rewriting inside $\delta(.)$ and possibly inside the scopes of quantifiers, case-splits are done in situ as the introduction of a $\vee$ on variables which would not be available in the hypothesis. We also attempt to keep the LHS's of wave-rules linear so that the only reason a match is prevented is occurrence of incompatible constructors. This helps to motivate relevant inductions. The rules in our glossary, appendix K, are in this form. Because we have context-sensitive rewriting and higher-order wave rules available the complicated RHS do not introduced many problems.

The `eval_def` proof follows closely that for `wave`, the equivalent wave rule rewriting tactic. We have also had success with a number of similar tactics. We noted the need for certain propositional wave rules and higher-order wave rules to be present in the above example. In our experience the same small set of such rules seems sufficient for most proofs.

4.5 Synthesis of compound tactics

We now look at how to put the tactics we have synthesized previously together, again verifying their correctness. We consider only the *verification* of compound tactics since the user is likely to have a given behaviour in mind and find it easier to express this procedurally (using a tactical) than with a complicated, monolithic specification. Of course, we could attempt to prove such a specification using the above of section 4.4.

Since our tactics are confined to being total we cannot specify those of the kind that would result from using the tactical `REPEAT`. (Howe, 1988a; Knoblock, 1987) point this problem out. They use the trick of replacing it by iteration a large number of times. Of course, such a construct will not be provably equivalent to `REPEAT`.

Suppose we have already synthesized `tac1` and `tac2`. Assume for simplicity that neither tactic has arguments. We therefore have proofs of:

$$\vdash \forall i : \texttt{metaterm.preconds1}(i) \rightarrow \exists o : \texttt{metaterm.effects1}(i, o) \wedge i \sim o$$
$$\vdash \forall i : \texttt{metaterm.preconds2}(i) \rightarrow \exists o : \texttt{metaterm.effects2}(i, o) \wedge i \sim o$$
$$\vdash \forall i : \texttt{metaterm.}\delta(\texttt{preconds1}(i))$$
$$\vdash \forall i : \texttt{metaterm.}\delta(\texttt{preconds2}(i))$$

We examine how we would put these together to form the specifications for supertactics.

`tac1 ORELSE tac2`

The synthesis and decidability theorems are:

$$\vdash \forall i : \texttt{metaterm.}(\texttt{preconds1}(i) \vee \texttt{preconds2}(i)) \rightarrow \exists o : \texttt{metaterm.}$$
$$((\texttt{preconds1}(i) \wedge \texttt{effects1}(i, o)) \vee (\neg\texttt{preconds1}(i) \wedge \texttt{effects2}(i, o))) \wedge i \sim o$$
$$\vdash \forall i : \texttt{metaterm.}\delta(\texttt{preconds1}(i) \vee \texttt{preconds2}(i))$$

and both are provable from the starting specifications.

TRY tac1

This is a degenerate case of ORELSE. It has the theorems:

$\vdash \forall i : \texttt{metaterm}.true \rightarrow \exists o : \texttt{metaterm}.$
$((\texttt{preconds1}(i) \wedge \texttt{effects1}(i, o)) \vee (\neg\texttt{preconds1}(i) \wedge i = oinmetaterm)) \wedge i \sim o$
$\vdash \forall i : \texttt{metaterm}.\delta(true)$

4.5.1 SUB tac1

SUB searches non-deterministically for a subterm of the input to which rewrite can apply. The result specification is:

$\vdash \forall i, x : \texttt{metaterm}, pos : \texttt{posn}.(\texttt{exp\_at}(i, pos, x) \wedge \texttt{preconds1}(x)) \rightarrow \exists o, k : \texttt{metaterm}.$
$\texttt{effects1}(x, k) \wedge \texttt{replace}(i, pos, k, o) \wedge i \sim o$
$\vdash \forall i : \texttt{metaterm}.\delta(\exists x : \texttt{metaterm}, pos : \texttt{posn}.\texttt{exp\_at}(i, pos, x) \wedge \texttt{preconds1}(x))$

Again, both of these are provable from the initial theorems.

PROGRESS tac1

By PROGRESS tac1 we mean that 1 has applied and the output is different from the input. The specifications are:

$\vdash \forall i.\texttt{preconds}(i) \rightarrow \exists o : \texttt{metaterm.effects1}(i, o) \wedge \neg i = o \texttt{ in metaterm} \wedge i \sim o$

tac1 THEN tac2

This has the specifications:

$\vdash \forall i, o2 : \texttt{metaterm}.(\texttt{preconds1}(i) \wedge \texttt{effects1}(i, o2) \wedge \texttt{preconds}(o2)) \rightarrow$
$\exists o : \texttt{metaterm.effects2}(o2, o)$
$\vdash \forall i : \texttt{metaterm}.\delta(\texttt{preconds1}(i) \wedge \exists o2 : \texttt{metaterm.effects}(i, o2) \wedge \texttt{preconds2}(o2))$

REPEAT tac1

Like all implementations of reflection in constructive type theory the representation of possibly non-terminating functions presents us with a problem. We cannot give a formal specification for REPEAT tac unless tac is of the most trivial nature, e.g. failtac. We show in chapter 5 a way around this: we allow a Prolog program to iterate (possibly forever) the compile pseudo-tactic for tac1. If this does terminate then we can count the iterations and apply the tactic formally that number of times. An important example of this type of tactic is the ripple of *CLAM*, defined as REPEAT wave. This method is appropriate there.

4.6 Decidability proofs

We shall distinguish in what follows between predicates and functions. In the type theory OYSTER we shall call any function whose result type is a universe (i.e. a type of the form Ui) a predicate; otherwise we call it simply a function. Since our type theory is constructive, all functions and predicates are total. This property is manifested internally in the constructive nature of the inference rules, and externally in the fact that the extract of a complete proof applied to appropriately typed objects evaluates to a normal form. However, our type theory is not decidable in general one cannot decide whether or not a given type is inhabited and so we have no algorithmic means of saying whether a given proposition is true or not. A class of propositions for which this is possible is those that are *decidable*, i.e. those $\mathcal{P}$ for which we can prove

$$\ldots \vdash \delta(\mathcal{P}) \tag{4.1}$$

Here we have used "decidable" in a different sense, applying to given propositions rather than theories as previously; context should make clear which sense is intended from now on. Typically $\mathcal{P}$ will have free variables and a theorem such as (4.1) provides us with a decision procedure for all instances of $\mathcal{P}$. This is useful in tactic synthesis since we often need to know when pre-conditions and effects hold

and in what cases the specification cannot be met. As OYSTER predicates are also used to model Prolog predicates, and we expect the latter to be computable, we should expect these to be decidable.

The decidability proof for a predicate allows us to extract much useful information. For example, if we are interested in synthesizing and using a predicate `p/2` in our proofs we would wish to have the following lemmas present:

$$\vdash \forall x, y.\delta(p(x, y))$$
$$\vdash \forall x.\delta(\exists y.p(x, y))$$
$$\vdash \forall y.\delta(\exists x.p(x, y))$$
$$\vdash \delta(\exists x, y.p(x, y))$$

These correspond respectively to use in modes (+,+), (+,-), (-,+) and (-,-). For each predicate of the *CLAM* method language we shall assume (and have proved) such lemmas. It will be shown below how these are used in synthesis proofs.

Our main tool in proving decidability theorems is a simple one but quite effective. We have the follow rules and their wave rule equivalents:

$$\begin{aligned}
\forall a, b : U1.\delta(a) \wedge \delta(b) &\rightarrow \delta(a \wedge b) \\
\forall a, b : U1.\delta(a) \wedge \delta(b) &\rightarrow \delta(a \vee b) \\
\forall a, b : U1.\delta(a) \wedge \delta(b) &\rightarrow \delta(a \rightarrow b) \\
\forall a, b : U1.\delta(b) &\rightarrow \delta(\neg b) \\
\texttt{true} &\rightarrow \delta(\texttt{true}) \\
\texttt{true} &\rightarrow \delta(\texttt{void})
\end{aligned}$$

As a demonstration of how general purpose rippling techniques can be applied, unchanged, to decidability goals we outline below the rippling story for the decidability of equality in arbitrary freely-generated parameterized data types. This is in contrast to the specific tactics developed for such goals by, for example, Hamilton (Hamilton, 1993) and Howe (Howe, 1988a). The only data required for the plan to succeed are the usual cancellation (wave) rules, uniqueness rules and the primitive induction schema which can be derived uniformly from the rules

for OYSTER's `rec` type. If the data type is parameterized over previously defined types we also need the lemmas asserting decidability for those types. We give as an example here our recursive type `metaterm`, and hope the reader can see how the proof would proceed in the general case.

We have the goal:

$$\vdash \forall x, y : \texttt{metaterm}.\delta(x = y \texttt{ in metaterm})$$

Individual induction on x and y is blocked since the cancellation rules are unable to ripple past the '='. Thus, simultaneous induction on x and y is suggested. We present three typical proof steps.

In the most complicated step case we have

$$\ldots \vdash \delta(\texttt{tbin}(f, a, b) = \texttt{tbin}(g, c, d) \texttt{ in metaterm})$$
$$\ldots \vdash \delta(f = g \texttt{ in atom} \wedge a = c \texttt{ in metaterm} \wedge b = d \texttt{ in metaterm})$$
$$\ldots \vdash \delta(f = g \texttt{ in atom}) \wedge \delta(a = c \texttt{ in metaterm} \wedge b = d \texttt{ in metaterm})$$
$$\ldots \vdash \delta(f = g \texttt{ in atom}) \wedge \delta(a = c \texttt{ in metaterm}) \wedge \delta(b = d \texttt{ in metaterm})$$
$$\ldots \vdash \delta(f = g \texttt{ in atom})$$

Here we successively used the cancellation wave rule, the wave rule for splitting up conjunctions inside $\delta(.)$, and finally strong fertilized. The last goal is assumed to be already present as a lemma.

When we have two non-identical recursive constructors the uniqueness lemmas are used to write it to the form $\delta(void)$, which is trivially true. Finally, when one of x and y is substituted for by a base constructor, the other is untouched. For example, one of the subgoals will be:

$$\ldots \vdash \delta(x = \texttt{tvar}(v) \texttt{ in metaterm})$$

The uniqueness rules suggest induction on x, which will end as in the first or second case depending on whether the constructors match.

We shall frequently use such decidability results below.

4.7 Context-sensitive rewriting

At various points in the foregoing example we needed more information during the rewriting of a term than was explicitly available from the hypothesis list. This is a not untypical case since tactic synthesis proofs are likely to contain many interacting subgoals. We have developed and implemented a partial solution to this problem by making use of as much of the surrounding goal as possible when rewriting a particular subgoal *context-sensitive rewriting (CSR)*. We define a function (see p. 85), by recursion over formulas containing a distinguished occurrence, which produces a set of sets of formulas the context which may be used as if they were part of the hypothesis list. We say *partial solution* because, as will be explained below, for reasons of tractability we limit the amount of inference which takes place in calculating a subterm's context. Below we use a presentation of typed, first-order intuitionistic predicate calculus derived from the Curry-Howard interpretation of OYSTER type theory. By dropping the uninteresting type declarations from sequents and the well-formedness subgoals from rules we arrive at the set of rules given in appendix B.

Our notion of context is based on an abstract notion of *fertilizability*. Suppose we have a goal of the form

$$\ldots, \mathcal{H}, \ldots \vdash \mathcal{G}[\mathcal{H}]$$

where $\mathcal{G}[\cdot]$ indicates a distinguished occurrence. We can rewrite the conclusion to $\mathcal{G}[\mathtt{true}]$ using the hypothesis $\mathcal{H}$ and stepping through $\mathcal{G}$ during rewriting (this is the process called strong fertilization in $CLAM$ when $\mathcal{H}$ is an induction hypothesis and $[\cdot]$ a wave hole). However, the instance of $\mathcal{H}$ we use to perform the rewriting may not be readily available explicitly as a hypothesis but may itself be buried in the goal $\mathcal{G}$. Consider a more concrete example:

$$\ldots \vdash \ldots((\mathtt{l} = \mathtt{nil\ in\ pnat\ list}) \# \mathtt{member(x, l)}) \ldots$$

where we also have available the rewrite

$$\forall x : \mathtt{T}, l : \mathtt{T}\ \ \mathtt{list}.(l = \mathtt{nil\ in}\ T\ \mathtt{list}) \rightarrow \mathtt{member}(x, l) \leftrightarrow \mathtt{void}$$

(For simplicity we will often refer to occurrences within a formula by using the names of meta-variables or object-level formulas which appear exactly once; "·" will often be used as an anonymous meta-variable.) For meta-level reasons, or if this goal were surrounded by other connectives for example, it may never happen that l = `nil` becomes a hypothesis. However, in the goal above l = `nil` *can* fertilize `member`(x, l) allowing it to be rewritten to `void`.

We generalise this example now to consideration of all formulas. Let us say that: an occurrence $[\cdot]$ in a formula $\mathcal{G}[\cdot]$ is fertilizable with respect to an occurrence $\mathcal{B}$ if any rewrite conditional upon $\mathcal{B}$ can validly be applied to $[\cdot]$. Since we are rewriting at the propositional level, i.e. using implications rather than equations, there is the usual notion of polarity involved (see, e.g., (Girard, 1987)). This leads us to formalise the above definition as:

Definition 4.1 (Fertilizability) *Occurrence $\mathcal{B}$ (strongly) positively fertilizes $[\cdot]$ in $\mathcal{G}[\cdot]$ iff*

$$\vdash (\forall.\mathcal{B} \rightarrow \underline{K \rightarrow A}) \rightarrow \mathcal{G}[K] \rightarrow \mathcal{G}[A]$$

Occurrence $\mathcal{B}$ (strongly) negatively fertilizes $[\cdot]$ in $\mathcal{G}[\cdot]$ iff

$$\vdash (\forall.\mathcal{B} \rightarrow \underline{A \rightarrow K}) \rightarrow \mathcal{G}[K] \rightarrow \mathcal{G}[A]$$

Occurrence $\mathcal{B}$ weakly fertilizes $[\cdot]$ in $\mathcal{G}[\cdot]$ iff

$$\vdash (\forall.\mathcal{B} \rightarrow \underline{K \leftrightarrow A}) \rightarrow \mathcal{G}[K] \rightarrow \mathcal{G}[A]$$

[Note that strong fertilizability of either kind implies weak fertilizability.] In these definitions K and A are distinct dummy propositional atoms which are assumed not to occur in $\mathcal{G}[\cdot]$, and whose free variables are exactly those of $\mathcal{B}$. $\forall$ is assumed to close everything in its scope. [The heuristic is to read these as: if $\mathcal{B}$ allows me to use the rewrite $A :\Rightarrow K$, what is the condition under which I can validly apply this, i.e. rewrite the goal $\mathcal{G}[A]$ to $\mathcal{G}[K]$?] Applying these definitions to the previous example we see that l = `nil` weakly fertilizes `member`(x, l) since, putting

$\mathcal{G}[\cdot] \equiv \ldots((l = \texttt{nil}_{\texttt{pnat list}})\#\cdot)\ldots$ and $\mathcal{B} \equiv l = \texttt{nil}_{\texttt{pnat list}}$, we can show that

$$\vdash (\forall x : \texttt{T}, l : \texttt{T list}.\mathcal{B} \rightarrow K(x,l) \leftrightarrow A(x,l)) \rightarrow \mathcal{G}[K(x,l)] \rightarrow \mathcal{G}[A(x,l)]$$

is a theorem. In the same way we see, as examples, that $\mathcal{B}$ positively fertilizes $\mathcal{A}$ in $\mathcal{B} \rightarrow \mathcal{A}$ and negatively fertilizes $\mathcal{A}$ in $(B \rightarrow A) \rightarrow C$. The idea of a context is to gather together all the $\mathcal{B}$'s w.r.t. a particular occurrence in a goal, and to treat these as if on the hypothesis list so that conditional rewriting at that occurrence is more likely to succeed.

The idea of using contexts when rewriting is a common one Boyer & Moore (Boyer & Moore, 1979) and NuPRL (Howe, 1988a; Constable *et al*, 1986) for example. Neither of these examples is appropriate to our requirements, however: Boyer & Moore, since they are working in quantifier-free, classical logic, are able to put all goals into a normal form, in essence moving everything that can be put there into a hypothesis list; NuPRL implements only a limited form of context sensitivity, described in the tactic documentation as "descending through conjunctions, through universal quantifiers, through implications (via the consequent only), and finally by applying ... a term destructor." This latter is not sufficient for our purposes as demonstrated by the examples given previously.

From the "semantic" definitions of fertilizability above it can be seen that an arbitrary amount of theorem proving may be necessary in order to decide the context of a term. We can make the context of an occurrence calculable in linear time by restricting the notion of context to one of *position only*, i.e. by considering each atomic proposition in the goal as distinct. Consider the goal

$$\vdash \texttt{C} \rightarrow (\texttt{C} \rightarrow \texttt{B}) \rightarrow \texttt{A}$$

According to our definition above A is positively fertilizable with respect to B, but this is by virtue of the inference $\vdash \texttt{C} \rightarrow (\texttt{C} \rightarrow \texttt{B}) \rightarrow \texttt{B}$. If we replace one of the C's by C', say, this is no longer the case. It is this restricted notion of context sensitivity, depending only upon the logical skeleton of the formula and not its contents, that we have implemented and explain below. We formalise this restricted notion of

context by applying the original definitions of fertilizability to a formula whose predicate symbols (including the OYSTER atom `void`) have been renamed so that no two are the same, e.g. by using indexes.

First let us define context. An occurrence $\mathcal{B}$ of a subformula of a (renamed) formula $\mathcal{G}[\cdot]$ is said to be in the context of $[\cdot]$ iff $\mathcal{B}$ weakly fertilizes $[\cdot]$. As is clear from the induction schema in the correctness proof below, the kind of fertilization (positive, negative or weak) depends only on the polarity of $[\cdot]$ within $\mathcal{G}[\cdot]$. Thus the context and polarity of $[\cdot]$ need only be calculated once. We now define a function `context` and show in the correctness proof that this computes the context as just defined. The last two cases are included for completeness though not primitive connectives.

$$
\begin{aligned}
\texttt{context}([\cdot]) &= \{\emptyset\} \\
\texttt{context}(\mathcal{C}[\cdot]\#\mathcal{D}) &= \{\{\mathcal{D}\}\} \otimes \texttt{context}(\mathcal{C}[\cdot]) + \text{symmetric case} \\
\texttt{context}(\mathcal{C}[\cdot]|\mathcal{D}) &= \texttt{context}(\mathcal{C}[\cdot]) + \text{symmetric case} \\
\texttt{context}(\mathcal{C} \rightarrow \mathcal{D}[\cdot]) &= \{\{\mathcal{C}\}\} \otimes \texttt{context}(\mathcal{D}[\cdot]) \\
\texttt{context}(\mathcal{C}[\cdot] \rightarrow \mathcal{D}) &= \texttt{context}(\mathcal{C}[\cdot]) \text{ where } \mathcal{D} \text{ is atomic} \\
\texttt{context}(\mathcal{C}[\cdot] \rightarrow \mathcal{P}\#\mathcal{Q}) &= \texttt{context}(\mathcal{C}[\cdot] \rightarrow \mathcal{P}) \cup \texttt{context}(\mathcal{C}[\cdot] \rightarrow \mathcal{Q}) \\
\texttt{context}(\mathcal{C}[\cdot] \rightarrow \mathcal{P}|\mathcal{Q}) &= \texttt{context}(\mathcal{C}[\cdot]) \\
\texttt{context}(\mathcal{C}[\cdot] \rightarrow \mathcal{P} \rightarrow \mathcal{Q}) &= \{\{\mathcal{P}\}\} \otimes \texttt{context}(\mathcal{C}[\cdot] \rightarrow \mathcal{Q}) \\
\texttt{context}(\mathcal{C}[\cdot] \rightarrow \forall x : T.\mathcal{D}) &= \texttt{context}(\mathcal{C}[\cdot] \rightarrow \mathcal{D}[y/x]) \text{ where } y \text{ is "fresh"} \\
\texttt{context}(\mathcal{C}[\cdot] \rightarrow \exists x : T.\mathcal{D}) &= \texttt{context}(\mathcal{C}[\cdot]) \\
\texttt{context}(\forall x : T.\mathcal{C}[\cdot]) &= \texttt{context}(\mathcal{C}[\cdot][y/x]) \text{ where } y \text{ is "fresh"} \\
\texttt{context}(\exists x : T.\mathcal{C}[\cdot]) &= \texttt{context}(\mathcal{C}[\cdot][y/x]) \text{ where } y \text{ is "fresh"} \\
\texttt{context}(\delta(\mathcal{C}[\cdot])) &= \texttt{context}(\mathcal{C}[\cdot]) \\
\texttt{context}(\mathcal{C}[\cdot] \leftrightarrow \mathcal{D}) &= \texttt{context}(\mathcal{C}[\cdot]) + \text{symmetric case}
\end{aligned}
$$

By $\otimes$ we mean here the set operation defined as:

$$A \otimes B = \{x \cup y | x \in A \wedge y \in B\}$$

4.7.1 Motivation for correctness of CSR

In the section below we prove the correctness of CSR, i.e. the theorem (in the positive case):

$$\Gamma, \mathcal{F}[K] \vdash \mathcal{F}[A] \iff \text{for all } S \in \texttt{context}(\mathcal{F}[\cdot]), \Gamma, S, K \vdash A$$

We can use this result to determine exactly when it is permissible to rewrite a goal of the form $\mathcal{F}[A]$ to $\mathcal{F}[K]$, i.e. when we can show $\mathcal{F}[K] \vdash \mathcal{F}[A]$. This is entailed by the correctness result as follows.

We first assume that we have a set of rewrite rules of the form $\mathcal{B}_i \to K \to A$ where $\mathcal{B}_i$ are atomic or the negations of an atoms. Putting Γ equal to this set of rewrite rules in the above gives us a necessary and sufficient condition for rewriting K to A. It remains to show that the RHS is true exactly when, for every $S \in \texttt{context}(\mathcal{F}[\cdot])$, there is some $\mathcal{B}_i \in S$.

Write Δ for any $S \in \texttt{context}(\mathcal{F}[\cdot])$. If $\mathcal{B}$ is a member of Δ then the RHS clearly holds. Now suppose, that the RHS holds, i.e. we have $\mathcal{B} \to K \to A, \Delta, K \vdash A$. Applying the interpolation theorem (4.2) with $\Gamma_1 = \Delta$, $\Gamma_2 = \mathcal{B} \to K \to A, K$, and $G = A$, we get an interpolant I all of whose predicate symbols (apart from void) are common to Δ and $\mathcal{B} \to K \to A$. This leaves only $\mathcal{B}$. This means I must be equivalent to one of the following: $\texttt{void}$, $\texttt{void} \to \texttt{void}$, $\mathcal{B}$, $\mathcal{B} \to \texttt{void}$, $(\mathcal{B} \to \texttt{void}) \to \texttt{void}$ (see section C.1 for proof). The interpolation theorem also tells us that

$\Delta \vdash I$ and $I, \mathcal{B} \to K \to A, K \vdash A$, and that where $\mathcal{B}$ does occur in I it occurs positively, thus eliminating the fourth case. Of the remaining cases all but the third can be eliminated by considering the first result. Thus we have $\Delta \vdash B$ and hence, by the form of Δ, that $\mathcal{B}$ is a member of Δ. It is this inexpensive test which forms the basis of our tactics and methods employing this result.

The procedure `context/5` does not explicitly carry around a sequent. Instead it notes the position (`Pos`) of the distinguished occurrence in the term (`Term`), the context so far (`Context`), and how many times `K` and `A` have swapped sides (`Switch` `+` for even; `-` for odd; `o` if we have both cases, e.g. δ, $\leftrightarrow$ or any truth-congruent connective rule). This simple form for `Switch` is possible since all the

rules are symmetrical in A and K. It will also be seen from the definition above that the number of operations required to calculate the context for a position in a formula is proportional to the depth of the position, and that if we allow for the breaking up of conjuncts and existentials that this is at most *linear* in the size of the formula (a proof of this result can be found in section C.6).

We extend the notion of context to include sequents using the strong conservativeness of the $\rightarrow$-intro rule, i.e. we treat a sequent $x_1 : h_1, \ldots, x_n : h_n \vdash G$ as $\vdash x_1 : h_1 \rightarrow \cdots \rightarrow x_n : h_n \rightarrow G$.

4.7.2 Correctness of context-sensitive rewriting

We work in a fragment of OYSTER type theory which is the equivalent of sorted first-order intuitionistic predicate calculus. Although we may be in the middle of a proof set in (and using rewrites based on) a much stronger theory typically one containing induction rules we assume that it is possible to carry out the rewriting process itself in this fragment. The presentation we use is given in appendix B. In this section, whenever we say a sequent is provable we mean provable in this fragment of the type theory. We also abuse the notion of provability in OYSTER slightly by saying that a sequent with a non-empty hypothesis list is provable by this we simply mean that there is a closed derivation which ends in that sequent. (Strictly speaking, for reasons of well-formedness, only sequents with empty hypothesis lists are provable in OYSTER. This will not be a problem for our fragment since well-formedness in it is a (decidable) syntactic property.) Before showing the correctness of context-sensitive rewriting we require some lemmas. By simple adaptations of standard proofs (for example those in (Girard, 1987)) we have shown the following for the OYSTER presentation of predicate calculus:

Theorem 4.1 (Cut-elimination) *If a sequent is provable then it is provable without using the* ***seq*** *rule.*

Theorem 4.2 (Interpolation theorem) *If the sequent* $\Gamma_1, \Gamma_2 \vdash G$ *is provable then there is a formula* I *(the interpolant) such that both* $\Gamma_1 \vdash I$ *and* $I, \Gamma_2 \vdash G$

are provable. Moreover, every predicate symbol (apart from **void***) that occurs in* I *occurs in both* Γ_1 *and* $\Gamma_2 \vdash G$ *with the same polarity; and if it occurs strictly positively in* I *then it occurs strictly positively in* Γ_1.

We use a corollary of this along with the fact that any formula whose only predicate symbol is `void` is either provable or its negation is. We have:

Theorem 4.3 (Corollary) *If the sequent* $\Gamma \vdash G$ *is provable but* Γ *and* G *have no proper (i.e. non-***void***) predicate symbols in common, then either:*

(i) $\Gamma \vdash$ **void** *is provable, or*

(ii) $\vdash G$ *is provable*

We also need three more proof theoretic lemmas. Each is proved in a similar fashion: induction on the cut-free proofs of the sequent mentioned in the assertion, with a case analysis of the last rule of such a deduction. Lemma 4.3 is used only in the proofs of lemmas 4.1 and 4.2.

Lemma 4.1 (Property A) *Suppose the sequent* $\Gamma \vdash \mathcal{C}|\mathcal{D}$ *is provable, and has in addition the following properties:*

1. *no symbol occurs more than once in* $\mathcal{D}$*, and* **void** *not at all*
2. *any symbols of* $\mathcal{D}$ *which appear in* Γ *occur only in the* $\mathcal{B}$ *parts in formulas of the form* $\forall \overline{x}.\mathcal{B} \to K \to A$ *or* $\forall \overline{x}.\mathcal{B} \to A \to K$
3. **void** *does not occur in* Γ *apart, possibly, from the* $\mathcal{B}$*'s of (2)*

Then $\Gamma \vdash \mathcal{C}$*. (An analogous result holds for* $\Gamma \vdash \mathcal{D}|\mathcal{C}$*.)*

Lemma 4.2 (Property B) *If the sequent* $\Gamma, C \to D \vdash D$ *is provable, no symbol occurs more than once in* D*, and* D *has no symbols in common with* Γ*, then* $\Gamma \vdash C$ *is provable.*

Lemma 4.3 (Property C) *Assume that hypothesis list* Γ *and formula* $\mathcal{D}$ *have properties (a)-(c) of lemma 4.1. Then* $\Gamma \not\vdash \mathcal{D}$.

Before stating the next theorem we must define a class of formulas commonly known as Harrop formulas (see (Smith, 1993) for historical details). We define this class inductively:

1. atomic formulas (including `void`) are Harrop
2. if A and B are Harrop then so is `A#B`
3. if A is Harrop then so is $\forall x : T.A$
4. if A is Harrop then so is $C \rightarrow A$ (for arbitrary C)

We can now state:

Lemma 4.4 (Generalized disjunction and existence properties) *Provided* Γ *contains only Harrop formulas:*

(i) if $\Gamma \vdash A|B$ *is provable then either* $\Gamma \vdash A$ *or* $\Gamma \vdash B$ *is provable.*

(ii) if $\Gamma \vdash \exists x : T.A$ *is provable then, for some term* t *of type* T, $\Gamma \vdash A[t/x]$ *is provable.*

Finally, we need to define the function * used to break up a context into constituent parts:

$$
\begin{array}{lcl}
A* & = & \{A\}\text{where } A \text{ is atomic} \\
(A \rightarrow B)* & = & \{A \rightarrow B\} \\
(A\#B)* & = & \{A\#B\} \cup A* \cup B* \\
(A|B)* & = & \{A|B\} \\
(\forall x : T.A)* & = & \{\forall x : T.A\} \\
(\exists x : T.A)* & = & \{\exists x : T.A\} \cup A*[y/x]\text{where } y \text{ is ``fresh''}
\end{array}
$$

We now give the promised informal proof showing the correctness of our context derivation procedure.

Theorem 4.4 (Correctness) *Suppose $\mathcal{G}[\cdot]$ is a formula with a distinguished occurrence as indicated. In order to limit our notion of context to one based on* position only *we now suppose that all atomic propositions (apart from* **void***) in $\mathcal{G}$ have been renamed so that all are distinct; and that boolean simplification has been applied so that* **void** *appears only as negation. Call this new goal $\mathcal{F}[\cdot]$. We can now apply the previous definition of abstract fertilizability to $\mathcal{F}$, viz. occurrence $\mathcal{B}$ positively fertilizes occurrence $[\cdot]$ with respect to $\mathcal{F}[\cdot]$ iff $\vdash (\mathcal{B} \to K \to A) \to \mathcal{F}[K] \to \mathcal{F}[A]$. The other definitions follow similarly. We now prove that $\mathcal{B}$ is a member of the context calculated by our procedure iff the above holds. I.e. that*

$$\Gamma, \mathcal{F}[K] \vdash \mathcal{F}[A] \iff \textit{for all } S \in \mathtt{context}(\mathcal{F}[\cdot]), \Gamma, S, K \vdash A$$

for positive $\mathcal{F}[\cdot]$, and

$$\Gamma, \mathcal{F}[K] \vdash \mathcal{F}[A] \iff \textit{for all } S \in \mathtt{context}(\mathcal{F}[\cdot]), \Gamma, S, A \vdash K$$

for negative $\mathcal{F}[\cdot]$, where Γ contains no propositional variables from $\mathcal{F}[\cdot]$, K or A except in rewrite rules of the form $\mathcal{D} \to K \to A$ or $\mathcal{D} \to A \to A$ where $\mathcal{D}$ is an atom or the negation of an atom.

We assume in the following that $[\cdot]$ is a positive occurrence in $\mathcal{C}[\cdot]$ and $\mathcal{D}[\cdot]$: the negative case has symmetric subproofs. We also assume that Γ contains no symbols in common with $\mathcal{F}[\cdot]$ apart from those in formulas of the form $\forall \overline{x}.\mathcal{B} \to A \to K$ or $\forall \overline{x}.\mathcal{B} \to K \to A$. Proof is by induction over formulas on $\mathcal{F}[\cdot]$ with the following case-split according to its outermost connective:

$\mathcal{F}[\cdot] \equiv \cdot$

We require $\Gamma, K \vdash A \iff$ for all $S \in \mathrm{context}([\cdot]), \Gamma, S, K \vdash A$. Since $\mathtt{context}([\cdot]) = \{\emptyset\}$ this is immediate.

$\mathcal{F}[\cdot] \equiv \mathcal{C}[\cdot]\#\mathcal{D}$

We require $\Gamma, \mathcal{C}[K]\#\mathcal{D} \vdash \mathcal{C}[A]\#\mathcal{D} \iff$ for all $S \in \mathtt{context}(\mathcal{C}[\cdot]\#D), \Gamma, S, K \vdash A$. By induction hypothesis we have $\Gamma', \mathcal{C}[K] \vdash \mathcal{C}[A] \iff$ for all $S' \in \mathtt{context}(\mathcal{C}[\cdot])$,

$\Gamma', K \vdash A$. Putting $\Gamma' = \Gamma, \mathcal{D}$ and $S = S', \mathcal{D}$ here, we see that is sufficient to show the equivalence of the LHS's. This is immediate since each direction corresponds to a simple derived rule.

Similarly for the case $\mathcal{F}[\cdot] \equiv \mathcal{C}\#\mathcal{D}[\cdot]$.

$\mathcal{F}[\cdot] \equiv \mathcal{C}[\cdot]|\mathcal{D}$

We require $\Gamma, \mathcal{C}[K]|\mathcal{D} \vdash \mathcal{C}[A]|\mathcal{D} \iff$ for all $S \in \mathtt{context}(\mathcal{C}[\cdot]|\mathcal{D})$, $\Gamma, K \vdash A$. By induction hypothesis we already have $\Gamma', \mathcal{C}[K] \vdash \mathcal{C}[A] \iff$ for all $S' \in \mathtt{context}(\mathcal{C}[\cdot])$, $\Gamma', K \vdash A$. Putting $\Gamma' = \Gamma$ and $S' = S$ we see that it is sufficient to show the equivalence of the LHS's. The $\Longleftarrow$ direction is immediate since this corresponds to a derived rule. The opposite direction is more problematic.

From $\Gamma, \mathcal{C}[K]|\mathcal{D} \vdash \mathcal{C}[A]|\mathcal{D}$ we immediately get $\Gamma, \mathcal{C}[K] \vdash \mathcal{C}[A]|\mathcal{D}$ via a derived rule. This now satisfies the syntactic conditions for using lemma 4.1, which gives us $\Gamma, \mathcal{C}[K] \vdash \mathcal{C}[A]$ as required.

Similarly for the case $\mathcal{F}[\cdot] \equiv \mathcal{C}|\mathcal{D}[\cdot]$

$\mathcal{F}[\cdot] \equiv \mathcal{C} \to \mathcal{D}[\cdot]$

We require $\Gamma, \mathcal{C} \to \mathcal{D}[K] \vdash \mathcal{C} \to \mathcal{D}[A] \iff$ for all $S \in \mathtt{context}(\mathcal{C} \to \mathcal{D}[\cdot])$, $\Gamma, K \vdash A$. By induction hypothesis we have $\Gamma', \mathcal{D}[K] \vdash \mathcal{D}[A] \iff$ for all $S' \in \mathtt{context}(\mathcal{D}[\cdot])$, $\Gamma', K \vdash A$. Putting $\Gamma' = \Gamma, \mathcal{C}$ and $S = S', \mathcal{C}$, we see that it is sufficient to prove the equivalence of the LHS's. This is immediate since both directions amount to simple derived rules.

$\mathcal{F}[\cdot] \equiv \mathcal{C}[\cdot] \to \mathcal{D}$ **where** $\mathcal{D}$ **is atomic**

We require $\Gamma, \mathcal{C}[K] \to \mathcal{D} \vdash \mathcal{C}[A] \to \mathcal{D} \iff$ for all $S \in \mathtt{context}(\mathcal{C}[\cdot] \to \mathcal{D})$, $\Gamma, A \vdash K$. By the induction hypothesis we have $\Gamma', \mathcal{C}[A] \vdash \mathcal{C}[K] \iff$ for all $S' \in \mathtt{context}(\mathcal{C}[\cdot])$, $\Gamma', A \vdash K$. Putting $\Gamma' = \Gamma$ and $S = S'$, it is sufficient to show the equivalence of LHS's. The $\Longleftarrow$ direction corresponds to a derived rule. The proof in the opposite direction results from lemma ??.

$\mathcal{F}[\cdot] \equiv \mathcal{C}[\cdot] \to \mathcal{D}\#\mathcal{E}$

We require $\Gamma, \mathcal{C}[K] \to \mathcal{D}\#\mathcal{E} \vdash \mathcal{C}[A] \to \mathcal{D}\#\mathcal{E} \iff$ for all $S \in \texttt{context}(\mathcal{C}[\cdot] \to \mathcal{D}\#\mathcal{E})$, $\Gamma, S, A \vdash K$. By induction hypothesis we have $\Gamma', \mathcal{C}[K] \to \mathcal{D} \vdash \mathcal{C}[A] \to \mathcal{D}$ $\iff$ for all $S' \in \texttt{context}(\mathcal{C}[\cdot] \to \mathcal{D})$, $\Gamma', S', A \vdash K$ and $\Gamma', \mathcal{C}[K] \to \mathcal{E} \vdash \mathcal{C}[A] \to \mathcal{E}$ $\iff$ for all $S'' \in \texttt{context}(\mathcal{C}[\cdot] \to \mathcal{E})$, $\Gamma', S'', A \vdash K$. Putting $\Gamma' = \Gamma$ and letting S range over the values for both S' and S'', we see that it is sufficient to show that the LHS of the required conclusion is equivalent to the conjunction of the LHS's of the induction hypotheses.

The $\Longleftarrow$ direction is immediate, corresponding to a derived rule. For the $\Longrightarrow$ direction: from the desired conclusion via a derived rule we have $\Gamma, \mathcal{C}[K] \to \mathcal{D}, \mathcal{C}[K] \to \mathcal{E} \vdash \mathcal{C}[A] \to \mathcal{D}$ and $\Gamma, \mathcal{C}[K] \to \mathcal{D}, \mathcal{C}[K] \to \mathcal{E} \vdash \mathcal{C}[A] \to \mathcal{E}$. From these, via lemma ??, we obtain the respective induction hypothesis LHS's.

$\mathcal{F}[\cdot] \equiv \mathcal{C}[\cdot] \to \mathcal{D}|\mathcal{E}$

The proof in this case is analogous to the case when $\mathcal{D}$ is atomic (see above).

$\mathcal{F}[\cdot] \equiv \mathcal{C}[\cdot] \to \mathcal{D} \to \mathcal{E}$

We require $\Gamma, \mathcal{C}[K] \to \mathcal{D} \to \mathcal{E} \vdash \mathcal{C}[A] \to \mathcal{D} \to \mathcal{E} \iff$ for all $S \in \texttt{context}(\mathcal{C}[\cdot] \to \mathcal{D} \to \mathcal{E})$, $\Gamma, A \vdash K$. By induction hypothesis we have $\Gamma', \mathcal{C}[K] \to \mathcal{E} \vdash \mathcal{C}[A] \to \mathcal{E}$ $\iff$ for all $S' \in \texttt{context}(\mathcal{C}[\cdot] \to \mathcal{E})$, $\Gamma', A \vdash K$. By putting $\Gamma = \Gamma, \mathcal{D}$ and $S = S', \mathcal{D}$, we see that it is sufficient to show the equivalence of the LHS's. This is immediate since both directions correspond to derived rules.

$\mathcal{F}[\cdot] \equiv \mathcal{C}[\cdot] \to \mathcal{D}$

Here we assume D is not of the form $P \to Q$. We require $\Gamma, C[K] \to D \vdash C[A] \to D \iff \Gamma, \mathit{context}(C[.] \to D), A \vdash K$. Note that on the RHS the K and A have swapped sides; this is the sole manifestation of polarity in this proof. By the induction hypothesis we have $\Gamma', C[A] \vdash C[K] \iff \Gamma', \mathit{context}(C[.]), A \vdash K$.

Putting $\Gamma' = \Gamma$, it is sufficient to show the equivalence of LHS's. The $\Longleftarrow$ direction corresponds to a derived rule. The proof in the opposite direction is as follows.

From $\Gamma, C[K] \rightarrow D \vdash C[A] \rightarrow D$ we get, via a derived rule, $\Gamma, C[K] \rightarrow D, C[A] \vdash D$. Write $\Gamma = B \rightarrow A \rightarrow K, \Gamma'$. There are two subcases:

(1) B has no symbols in common with D. By theorem 4.2 we have an interpolant I which contains no symbols from D and such that $B \rightarrow A \rightarrow K, \Gamma', C[A] \vdash I$ and $I, C[K] \rightarrow D \vdash D$. By lemma 4.2 we have $I \vdash C[K]$, from which the desired result follows.

(2) B has a symbol in common with D. By theorem 4.2 we have an interpolant I containing only symbols common to B and D and such that $B \rightarrow A \rightarrow K, \Gamma' \vdash I$ and $I, C[A], C[K] \rightarrow D \vdash D$. Note that K and A cannot occur in I since they have the wrong polarities and that any symbol which occurs in I occurs with opposite polarity to its occurrence in B. Apply the interpolation theorem again to the first sequent we get a J containing only the symbol `void` and such that $\Gamma', B \rightarrow A \rightarrow K \vdash J$ and $J \vdash I$. Since J must be equivalent to true or false either $\Gamma', B \rightarrow A \rightarrow K \vdash$ `void`, which is impossible because of the syntactic restrictions on Γ; or $\vdash I$, whence $C[A], C[K] \rightarrow D \vdash D$ and $C[A] \vdash C[K]$ by lemma 4.2.

QED

4.7.3 Elimination of quantifiers

We remarked in the course of the example in section 4.3 that it is often possible to eliminate a quantifier if in its scope there is an equality between the variable it binds and some other term. This was motivated by the following type theoretic equivalences, where t is an arbitrary term of type T:

$$(\forall x : T.x = t \text{ in } T \rightarrow C) \leftrightarrow C[t/x]$$

$$(\exists x : T.x = t \text{ in } T \wedge C) \leftrightarrow C[t/x]$$

Since we have the OYSTER theorems:

$$\vdash \forall T : \mathtt{U1}.\forall t : T.\forall C : (T \rightarrow \mathrm{U2}).(\forall x : T.x = t \text{ in } T \rightarrow C(x)) \leftrightarrow C(t)$$

$$\vdash \forall T : \mathtt{U1}.\forall t : T.\forall C : (T \rightarrow \mathrm{U2}).(\exists x : T.x = t \text{ in } T \# C(x)) \leftrightarrow C(t)$$

in order to use these equivalences in an actual proof we need to breakdown a formula in the same manner as in the proof above and, at the point when C[.] reaches the outside and by which point we will have appropriately typed T and t, show that:

$$\ldots \vdash \lambda x.C \text{ in } (T \rightarrow \mathrm{U2})$$

But this would have arisen as part of the well-formedness subproof of the original formula. [Since they are sufficient for our work we have used universes U1 for data types and U2 for propositions here; analogous theorems hold for arbitrary universes.]

We also noted that in order to eliminate the quantifier, for soundness certain criteria had to be met on the position of the equality. Having described the notion of context-sensitive rewriting, and with the above equivalences in mind, one possible set of criteria becomes apparent. Let us look at the universal case. Suppose we wish to eliminate the outermost quantifier in the formula $\forall x : T.\mathcal{D}[x = t \text{ in } T]$, where $\mathcal{D}[\cdot]$ again indicates a distinguished occurrence. Let $*$ be another anonymous meta-variable. If $\cdot$ is in the context of $*$ in the formula $* \rightarrow \mathcal{D}[\cdot]$ then we have the following simple equivalence-preserving sequence of rewritings:

$$\forall x : T.\mathcal{D}[x = t \text{ in } T] \tag{4.2}$$

$$\forall x : T.\ \mathtt{true} \rightarrow \mathcal{D}[x = t \text{ in } T] \tag{4.3}$$

$$\forall x : T.(x = t \text{ in } T) \rightarrow \mathcal{D}[x = t \text{ in } T] \tag{4.4}$$

$$\mathcal{D}[x = t \text{ in } T][t/x] \tag{4.5}$$

$$\mathcal{D}[\mathtt{true}][t/x] \tag{4.6}$$

Line 4.3 logically implies line 4.4. Line 4.4 implies line 4.3 by virtue of CSR with the conditional rewrite $(x = t \text{ in } T) \rightarrow (x = t \text{ in } T) \leftrightarrow \mathtt{true}$ since we are given that $\cdot$ lies in the context of $*$. A similar situation holds for the existential case:

$$\exists x : T.\mathcal{D}[x = t \text{ in } T]$$

$$\exists x : T.\ \mathtt{true} \wedge \mathcal{D}[x = t \text{ in } T]$$

$$\exists x : T.(x = t \text{ in } T) \wedge \mathcal{D}[x = t \text{ in } T]$$

$$\mathcal{D}[x = t \text{ in } T][t/x]$$

$\mathcal{D}[\texttt{true}][t/x]$

Thus our criteria on the position of the equalities are:

1. In the case of a universal quantifier: if $\cdot$ is in the context of $*$ in the formula $* \rightarrow \mathcal{D}[\cdot]$ then one may replace $\forall x : T.\mathcal{D}[x = t \texttt{ in } T]$ with $\mathcal{D}[\texttt{true}][t/x]$.

2. In the case of an existential quantifier: if $\cdot$ is in the context of $*$ in the formula $* \wedge \mathcal{D}[\cdot]$ then one may replace $\exists x : T.\mathcal{D}[x = t \texttt{ in } T]$ with $\mathcal{D}[\texttt{true}][t/x]$.

Note: a variable may not be explicitly quantified in the goal but instead declared in the hypothesis list. The same criterion applies here as for universal quantifiers with the difference that we thin the declaration rather than eliminate a quantifier. This is the case that is handled by *CLAM*'s existing `equal/2` submethod.

In inductive parts of proof we need to ensure that applying these elimination procedures does not disrupt the skeleton. However, they are positively useful if the equalities occur in a wave front since they shrink or eliminate that front. For this reason this submethod has been made part of the unblocking submethod, `unblock/3`. In addition we eliminate any quantifier which, because of earlier rewriting, no longer has a free occurrence of its variable in its scope.

These criteria are certainly not the most complete since they depend on the notion of context-sensitive rewriting and, as noted above, this is based on position only. However, we have found it very useful in practice. These criteria also have the virtue of being decidable with linear complexity. These features have lead us to use rewrite rules which feature equalities on the right-hand side rather than several conditional rules.

4.8 Higher-order wave rules

The rippling process in the example given above became blocked where the wave front reached a quantifier. This is to be expected since *CLAM* uses only first-order matching for wave rules at present. We had a goal of the form:

$$\vdash \forall x : t. \boxed{p \wedge \underline{q(x)}} \quad (1)$$

where p did not contain x free. It is clearly valid to perform the desired ripple to

$$\vdash \boxed{p \wedge \underline{\forall x : t. q(x)}}$$

but the current formulation of wave rules cannot soundly handle such cases: its first-order matching is incapable of expressing variable capture. In order to be able to perform such ripples we have at least three choices:

- Use first-order rules but allow meta side-conditions to test for variable capture, perform α-conversion. This would require careful entry of wave rules and be opaque to the user.

- extend the unblocking method to handle such cases explicitly. This is extremely ad hoc and cannot served as a principled way to store rewrite rules.

- extend the notion of wave rule to cover such cases. This would require higher-order unification to take account of binding constructors.

For reasons of being able to tell a coherent "rippling story", being able to write transparently correct wave rules and of retaining more natural proof plans we have chosen the last option. We shall see below that this brings with it unexpected advantages and few disadvantages.

We consider here only wave rules where all the wave fronts contain at most one hole, i.e. where rippling is aiming towards a single hypothesis. We discuss at the end of this section how the notion might be extended to multi-hole rules. Our idea is to annotate waves using a binding construct (Liang, 1992) in the abstract

syntax of OYSTER and express the LHS of wave rules as higher-order patterns (hop's) (Miller, 1991; Nipkow, 1991)[1]. Our novel idea is to include higher-order variables expressing the presence of potential wave fronts in all wave holes. The result is still a hop. We first give some preliminaries.

Waves are used to distinguish subterms of a goal, and within each such subterm a further subterm. The notation is used to make explicit the difference between a particular hypothesis and the goal, the latter containing extra structure which we hope to ripple out to obtain a match. As shown by Liang (Liang, 1992), we can elegantly represent wave fronts using a binding construct, which we shall call `wave`. The wave front $t[\boxed{c(\underline{x})}]$ is expressed as $t[(wave(\lambda y.c(y))x)]$. Using such a notation it is easy to supply a semantics for the wave notation. We consider the two δ-rules:

$$\texttt{(wave C X)} \quad \text{evaluates to} \quad \texttt{(C X)} \quad (\delta 1)$$
$$\texttt{(wave C X)} \quad \text{evaluates to} \quad \texttt{X} \quad (\delta 2)$$

The expression with all wave fronts removed (the erasure) is got by evaluating using ($\delta 1$), the skeleton using ($\delta 2$). Hence our semantics: we require that wave rules preserve the skeleton and express a given relation between erasures. If the wave rule is $L :\Rightarrow R$ and the relation is $\sim$ we have:

$$\texttt{skeleton}(L) = \texttt{skeleton}(R)$$
$$\texttt{erase}(L) \sim \texttt{erase}(R)$$

In choosing to use this notation we are faced with the choice of how to implement it. We can either extend the syntax of the object logic or use a meta annotation. The former would allow the possibility of confusing object- and meta-level abstraction and application, so instead we chose to extend the underlying abstract syntax of OYSTER. Although neither explicitly uses an abstract syntax they are both based on Martin-Lof type theory (Martin-Löf, 1984) where a *theory of expressions* is given. It is a curried version of this that we extend with the binding `wave` constructor.

---

[1]For reasons of self-containment a brief overview and explanation of terminology can be found in section 6.7.

[In what follows we use the Prolog convention of giving variables names beginning with a capital, and constants lower case letters.] We will use the notation $x\backslash\backslash M$ for abstraction and `M @ N` for application. So, for example, the object level terms $\lambda x.M$, `P of x`, `plus(x,y)` and $\forall x : t.P$ have the abstract syntax respectively `λ @ (x \\M)`, `of @ P @ x`, `plus @ x @ y` and `∀ @ t @ (x \\P)`.

We can now immediately express higher-order wave rules such as (1).

```
∀ @ T @ (x \\wave @
   :⇒
wave @ (z \\∧ @ P @
```

[In fact this is not quite true: see section 4.11.4 for details.] By using higher-order matching, and since `P` does not have `x` as an explicit argument, we can ensure that the meta condition that `P` does not have `x` free is observed.

After studying a large number of wave rules it was noticed that the LHS always has the form of a higher-order pattern (Miller, 1991), i.e. free variables only appear in the form `F @ x1 @ ... @ xn`, where `xi` are distinct bound variables. This is not surprising since single hole wave rules have the general form

$$f[\![\boxed{ci(\underline{Xi})}]\!] :\Rightarrow \boxed{w(\underline{f[Xi]})}, \text{ i.e.}$$
$$f[(wave@ci@Xi)] :\Rightarrow (wave@w@f[Xi])$$

Miller shows that, unlike full higher-order unification, unification between hops is decidable. Further, it has the other desirable properties of first-order matching of having a most general unifier when unifiers exist (Miller, 1991) and having a linear algorithm (Qian, 1992). We propose limiting higher-order wave rules to having hops for LHS's.

The form of the wave rule given above is not as flexible as it could be: it requires that the wave holes in the LHS match exactly with those of the goal. This is not always the case, even when the rule is applicable. The reason for this is that a wave front may itself fill a wave hole. [This has led to the use of split and join operations on wave fronts and a normal form where each wave front is exactly one functor thick.] Consider a goal containing the wave front $\boxed{s(\underline{s(x)})}$ and a wave rule with LHS $\boxed{s(\underline{X})}$. These only match after splitting the goal wave front to give $\boxed{s(\boxed{\underline{s(x)}})}$, with the result that X is bound to $\boxed{s(\underline{x})}$. The higher-order notation

allows us to match such wave fronts *directly*. We do this by including in the wave rule LHS a new variable which "absorbs" any wave front in the wave hole. The LHS wave front would thus be written

`(wave @ (x \\s @ (D @ x)) @ X)`,

where `D` is our new potential wave front variable. Since `D` may become instantiated with a wave front we now have to replace all occurrences of `X` in the RHS with `(wave @ D @ X)`. If the wave fronts match exactly then `D` is unified with `x \\x` and we may remove the degenerate front.

Thus, we now think of every wave hole in the LHS of a rule as containing a potential wave, i.e. we now have:

$$f[\boxed{ci(\boxed{Xi})}] :\Rightarrow \boxed{w(f[\boxed{Xi}])}, \text{ i.e.}$$

$$f[(wave@(x\backslash\backslash ci@(Di@x))@Xi)] :\Rightarrow (wave@w@f[(wave@Di@Xi)])$$

Note that these are still hops. The translation of general theorems into such wave rules is easily automated, using existing algorithms for difference matching and uniformly adjusting the wave holes as shown. The soundness of this approach to rippling follows immediately from the soundness of unification and the fact that wave rules are taken to preserve skeletons and equivalences between erasures. We also assume that the β-normal form of the RHS contains no variables which do not appear in β-normal form of the LHS: this ensures that rippling introduces no uninstantiated terms.

Theorem 4.5 *Correctness*

If rewriting is carried out as described above then the result is correct. I.e.:

Suppose that in the goal $G[X]$ we rewrite the subterm `X` using the h.o. wave rule $L :\Rightarrow R$ for the relation $\sim$. Suppose also that hop unification returns the unifier θ s.t. $\theta L = X$ and that the goal is rewritten to $G[\theta R]$. Then by correctness we mean:

$$skeleton(G) = skeleton(G[\theta R])$$
$$\vdash erase(G[\theta R]) \rightarrow erase(G)$$

In the above we assume that $G[X]$, L and R are (monohole wave) hops annotated using the Liang notation. We additionally assume that the relation $\sim$ is such that

$A \sim B$ implies $\theta A \sim \theta B$ for all substitutions θ and terms A and B; and that $\sim$ and `G[.]` are such that, for all wave free terms `X, Y`

`X ~ Y` implies `erase(G[Y]) → erase(G[X])`.

Proof: From the definition (δ2) and a simple induction on the abstract syntax we have

`θ (skeleton(T)) = skeleton(θ T)`.

From property (1) of a wave rule it follows that `skeleton( θ L) = skeleton( θ R)`.

Similarly we have

`θ (erase(T)) = erase( θ T)`.

From the first assumption we have:

`erase( θ L) ~ erase( θ R)`.

Requirement (1) now follows from (3) and the easily proved (as for (1)) fact that `skeleton(X) = skeleton(Y)` implies `skeleton(G[X])=skeleton(G[Y])`.

From the second assumption, and noting that `erase` is idempotent, we now have, from (4), that $\vdash erase(G[erase(\theta L)]) \rightarrow erase(G[erase(\theta R)])$
which is our 2nd correctness requirement.

QED

Note: for `G[ θ R]` to be syntactically correct we require that `θ R` is ground. This is ensured by our condition on the appearance of variables in wave rules.

We now note some unexpected advantages arising from this formulation.

1. We no longer need to split wave fronts in order for wave rules to apply as was previously the case. The normal form is now the much simpler and space-efficient single wave front form. There is no need to explicitly put terms into this form since the join rule is itself a higher-order wave rule (as noted by (Liang, 1992)):

```
wave @ F @ (wave @ G @ X) :⇒
wave @ ((f \\g \\x \\f @ g @ x) @ F @ G) @ X
```

and thus we no longer need explicitly join wave fronts either.

2. The presence of the potential wave front `Di` allows a least-commitment form of middle-out reasoning (MOR) (Bundy *et al.* 1989a) where, even after applying several wave rules which instantiate the wave front around the induction variable, any number of further ripples are still possible. We start the search with the induction variable, say `x`, replaced with `wave @ C @ x`. We know that fertilization is possible when the wave front next becomes a pure variable. Of course, it is still necessary to show that the eventual instantiation of C results in a well-founded order for the induction.

3. Being able to use h.o. wave rules will extend the domain to which CL^AM can be applied. We can reason about higher-order abstract programs, e.g. fold and map, in a way not previously possible.

Some possible disadvantages of using higher-order wave rules are:

1. The introduction of extra `wave` terms. This is not a problem at all since these only appear when necessary for a wave rule to apply. In the degenerate case, i.e. when the wave front is `x \\x`, they are easily removed. We have implemented this by adding, at a high-priority, the wave rule

$$\texttt{(wave @ (x \\x) X)} :\Rightarrow X$$

In MOR, pure speculative fronts are easily distinguished as by the presence of a pure Prolog variable as the wave front.

2. The expense of higher-order unification. As stated above, a linear algorithm is know for hop unification (Qian, 1992). In addition the use of binding constructors need not be expensive: the raison d'etre of hops is that normalization is possible using only $\beta_0\eta$-reduction which is very cheap. As with f.o. wave rules, we may index h.o. wave rules by, for example, outermost functor and outermost wave functor ensuring that only a small number of unifications are attempted. Further, it is possible to hard-wire a highly efficient C implementation of the Qian algorithm into Prolog for use as a compiled predicate.

4.8.1 Multi-hole wave rules

We have examined the problem of extending the above formalism to multi-hole wave fronts, i.e. front where each wave hole pertains to a distinct hypothesis. The only immediate way forward would appear to extend the abstract syntax with a countable infinity of new constructs `waveN` corresponding to N-hole wave fronts (`wave1` is our original `wave`). `waveN` has type $(o \rightarrow \ldots \rightarrow o) \rightarrow o \rightarrow \ldots \rightarrow o \rightarrow o$, where o is the type of saturated expressions, so that `waveN` has the wave front as its first argument and the ith wave hole as its (i+1)th argument. For example we would express

$\boxed{plus\underline{a_1}\underline{b_2}}$ as

```
wave2 @ (x \\y \\plus @ x @ y) @ a @ b
```

The most serious shortcoming of this notation is that wave holes are indexed by their position in the term, i.e. their argument number. We cannot, therefore, use unification to match wave hole indices, and require instead that an N-hole wave rule be present in all its N! permutations. The final example demonstrates this problem. This is not too burdensome for small N in practice we have not encountered N greater than 2.

Our use of an absorbing wave front variable is straightforwardly extended to the multi-hole case. As before each binding occurrence of a hole, say `xi`, is replaced by `Di @ xi`, and each occurrence of `Xi` on the RHS is replaced by `wave @ Di @ Xi`. We will use the multi-hole rule

$$\boxed{\underline{U_1} + \underline{V_2}} = \boxed{\underline{X_1} + \underline{Y_2}} :\Rightarrow \boxed{\underline{U = X_1} \wedge \underline{V = Y_2}}$$

as an example. Compiled into our formalism we have the two permutations:

```
l eq @ (wave2 @ (x \\y \\plus @ (C1 @ x) @ (C2 @ y)) @ U @ V)
    @ (wave2 @ (x \\y \\plus @ (D1 @ x) @ (D2 @ y)) @ X @ Y) :=>
wave2 @ (x \\y \\^ @ x @ y) @ (eq @ (wave @ C1 @ U) @ (wave @ D1 @ X))
                            @ (eq @ (wave @ C2 @ V) @ (wave @ D2 @ Y))
```

```
eq @ (wave2 @ (x \\y \\plus @ (C2 @ y) @ (C1 @ x)) @ U @ V)
```

@ (wave2 @ (x \\y \\plus @ (D2 @ y) @ (D1 @ x)) @ X @ Y) :⇒
wave2 @ (x \\y \\∧ @ y @ x) @ (eq @ (wave @ C1 @ U) @ (wave @ D1 @ X))
@ (eq @ (wave @ C2 @ V) @ (wave @ D2 @ Y))

4.8.2 Wave terms

In the following we consider only monochromatic, mono-hole wave annotation it should be quite straight forward to extend the idea to the polychromatic/multi-hole case by extending the skeleton and erase functions. We try to take as abstract a view of wave terms and their properties as possible so as not to preclude some forms of representation. For this reason we give a specification in terms of the extensional properties (essentially observable properties) rather than intensional (in terms of annotation is the box/hole representation already an over-specification?). The essential properties of a wave term are that it allows access to something called the *erase* part, something called the *skeleton* and that these are related by a special subterm relation. We also need to specify how wave terms are affected by substitution. Thus:

A **wave term** is a data structure with at least the selectors `erase` and `skeleton`. These are bound by the condition that for all wave terms `t`, `skeleton(t)` is a *wave-subterm* of `erase(t)`. The wave-subterm relation is an extension of the usual subterm one which allows deletions. More precisely:

$f(s_1, \ldots, s_m)$ is a wave-subterm of $g(t_1, \ldots, t_n)$ iff
(i) $f = g, m = n$ and $\forall i.s_i$ is a wave-subterm of t_i; or
(ii) $\exists i.f(s_1, \ldots, s_m)$ is a wave-subterm of t_i.

[Note: strange though it may seem, this condition does not appear to be required for the definition of wave-rewriting which follows.]

We also need to specify the effect of a substitution on a wave term. Using the abbreviation $f(\sigma)$ for $\{< v, f(z) > \mid < v, z > \epsilon\sigma\}$ we require that
(i) skeleton(σt) = (skeleton(σ))(skeleton(t)), and
(ii) erase(σt) = (erase(σ))(erase(t)).

From these definitions it may be inferred that we allow wave terms as well as normal terms to appear in the right-hand sides of substitutions.

[Note: We have required that the skeleton of a wave term always be defined. This leads to a slight difficulty if our definition is to include hole-less wave terms. This can be patched since the skeleton of a hole-less wave is immaterial: we can satisfy the wave subterm condition by setting the skeleton to be some special null term which is a wave-subterm of every term.]

[Note: we can regard an ordinary term t as a degenerate wave term t', i.e. one in which $t = erase(t') = skeleton(t')$.]

4.9 Wave rewriting

Since what are traditionally called wave rules are derived from ordinary equations/formulas by performing a parsing operation, and since we have found it necessary in practice to "jiggle" with the results (e.g. allow weakening of a rule), we define wave rewriting with reference to the original equation/formula rather than a wave rule — i.e. we perform on-the-fly wave rule parsing.

We suppose we have some kind of rule $L \sim R$ which is universally closed. [We do not consider here unification under a mixed prefix.] We define wave rewriting of a term $t[s]$ at the subterm s as follows:
(i) There exist wave terms L' and R' such that $erase(L') = L$ and $erase(R') = R$, and $skeleton(L') = skeleton(R')$
(ii) There exists a substitution σ s.t. $erase(s) = erase(\sigma L')$ and $skeleton(s) = skeleton(\sigma L')$.

The result is the (wave) term $t[\sigma R']$.

The motivation behind wave rewriting is to preserve some kind of equivalence and to preserve skeletons (and, of course, to move "closer" to some desired term). We show that this is in fact the case if the above conditions are met.

$skeleton(\sigma R') = (skeleton(\sigma))(skeleton(R')) = (skeleton(\sigma))(skeleton(L')) = skeleton(\sigma L') = skeleton(s)$

Similarly, we can show $erase(s) \sim erase(\sigma R')$ which is the equivalence required.

The difficult part in the definition is finding L' and R' given L and R, though clearly, according to our earlier definition of wave term we can exhaustively try all wave-subterms of L as the skeleton: suitable ones will be those which are also wave subterms of R and which satisfy the match conditions (ii). We have also not defined what it means to be a subterm of a wave term or to replace such a subterm with another, i.e. we haven't defined what $t[s]$ and $t[\sigma R']$ mean.

4.10 A wave term language?

In section 1 it was suggested that we could think of normal terms as degenerate wave terms. If this is to be of any use we must describe how one can perform the usual linguistic operations of abstraction and application on wave terms and what an atomic wave term is? In keeping with the rest of this section these descriptions should be semantically based and independent of any particular notation.

We now propose a wave calculus based on the normal applicative/abstractive structure of the lambda calculus. We write wave terms explicitly as pairs, $< erasure, skeleton >$, and show how these can are combined to cover the whole language. We start from the semantic definition of what it is to be a wave subterm in a lambda-calculus type language. It will be noted that we lose the "pleasant" property of always having the erasure and skeleton be of the same type.

s is a wave-subterm of t iff one of the following is the case:

1. s and t are identical atoms.

2. s is AB and t is UV and A is a wave-subterm of U and B is a wave-subterm of V.

3. t is UV and s is a wave-subterm of U.

4. t is UV and s is a wave-subterm of V.

5. s is $\lambda x.A$ and t is $\lambda x.B$ and A is a wave-subterm of B.

6. t is $\lambda x.A$, s does not contain x free and is a wave-subterm of A.

[It is cases 3 and 6 which no longer preserve the box/hole type match.] This definition motivates our definition of a wave language. Note that there are now 3 kinds of application and 2 kinds of abstraction. Note that we have the property that non-annotated terms t can be simply injected into the wave terms as $< t, t >$.

1. $< u, u >$ is a wave term, where u is an atom.

2. $< a, b > \otimes < u, v >$ is the wave term $< au, bv >$.

3. $< a, b > \otimes_1 < u, v >$ is the wave term $< au, b >$.

4. $< a, b > \otimes_2 < u, v >$ is the wave term $< au, v >$.

5. $\lambda x. < a, b >$ is the wave term $< \lambda x.a, \lambda x.b >$.

6. $\lambda' x. < a, b >$ is the wave term $< \lambda x.a, b >$, where b does not contain x free.

Let us give some examples. The wave term $\boxed{s(\underline{s(x)})}$ can be written $< s, s > \otimes_2 (< s, s > \otimes < x, x >)$, and the wave term $s(\boxed{s(\underline{x})})$ as $< s, s > \otimes (< s, s > \otimes_2 < x, x >)$. Since all terminals have the form $< u, u >$ where u is an atom, we may abbreviate accordingly.

Does this wave term calculus allow us to describe $t[.]$? Does it help simplify the unification of annotated terms?

4.11 Problems

4.11.1

Are the properties we have chosen above to characterize annotated terms, viz. erasure, skeleton and substitution behaviour, sufficient for a characterization? Consider the wave term $< (x+x)+(x+x), x+x >$. There are six different annotations

that could have give rise to this:

$\boxed{\underline{(x+x)}+(x+x)}$

$\boxed{(x+x)+\underline{(x+x)}}$

$\boxed{(\underline{x}+x)}+\boxed{(\underline{x}+x)}$

$\boxed{(\underline{x}+x)}+\boxed{(x+\underline{x})}$

$\boxed{(x+\underline{x})}+\boxed{(\underline{x}+x)}$

$\boxed{(x+\underline{x})}+\boxed{(x+\underline{x})}$

Do they all really mean the same? Now consider the effects of rewriting these. Simple wave rewriting would not appear to be a problem since only the erasure is affected. [In the higher-order case we will have to consider $\alpha\beta\eta-$conversion as well.] What is more interesting is ordinary rewriting. This is restricted in Clam so that it never changes the skeleton (therefore possibly upsetting a later fertilization), but we can't immediately be sure how the skeleton and erasure are related by looking at the wave term pair. If, for example, we used the (admittedly strange) equality $x + x = a$, then it would be applicable to the first 2 cases but not the last 4. In either case the Clam restriction ensures that the skeleton is preserved so this would also appear to be OK. How could we tell, though, just from the wave term pair, that it could not be used at the top level to give $< a, x + x >$? Is it sufficient merely to check that the result of rewriting is still a wave term? In that case: what is the difference between a wave-rule and an ordinary rewrite?

4.11.2 Counter-examples

To counter the thought that we can get away with wave-rewriting merely by checking that the erasure of the rule is a subterm of the erasure of the goal and that (with the resulting substitution applied) the skeleton of the rule is a subterm of the skeleton of the goal, we have the following. Consider the goal $\boxed{f(\underline{a},c)}+c$ (i.e. $< f(a,c)+c, a+c >$) to which the wave-rule $\boxed{f(a,\underline{c})} :\Rightarrow c$ (i.e. $< f(a,c), c >:\Rightarrow< c, c >$) is to be applied. Although one can see immediately from the notation that there is no wave-match (whatever that means), the false hypothesis mentioned above does indeed hold and would result in the rewritten

goal $< c + c, a + c >$ which is not a wave-term. It is clear what is going on here: the occurrence of c inside the skeleton is being confused with the one which is not.

We can also give a counter-example to show that naive rippling is non-terminating. We have the two wave-rules:

$$\boxed{\underline{X} + c} + c \;:\Rightarrow\; \boxed{\underline{X + c} + d} \tag{4.7}$$

$$\boxed{\underline{X} + d} + c \;:\Rightarrow\; \boxed{\underline{X + c} + c} \tag{4.8}$$

Starting from the goal $(\boxed{\underline{a} + c} + c) + c$ we get the vicious cycle: $(\boxed{\underline{a} + c}) + c :\Rightarrow \boxed{\underline{a + c} + d} + c :\Rightarrow \boxed{\underline{(a + c) + c} + c} :\Rightarrow (\boxed{\underline{a} + c}) + c$. The last step comes about since we can't distinguish the two from only their erasures and skeletons.

4.11.3

What is the unification algorithm for the wave term calculus like? For example, by virtue of the 3 kinds of application and 2 kinds of abstraction there are 6 kinds each of β- and η-conversion, not all of which appear to be meaningful.

4.11.4 Aspects of implementation

Higher-order wave rules have been implemented for OYSTER almost exactly as stated above. We have used the unification algorithm for hops given by Nipkow (Nipkow, 1991) in such a form that Prolog variables correspond to higher-order free variables. This leads to a small problem in that care must be taken to avoid variable capture as Prolog variables become instantiated. This can be circumvented by rewriting all occurrences of Prolog variables that are in the scope of a \\ binding in the form of an application. For example, in the RHS's of wave rules instead of writing

```
(x \\g @ F)
```

we would write

```
(y \\x \\g @ y) @ F
```

and allow $\beta_0\eta$-reduction to perform the substitution soundly. This explains the rather long-winded form of the join rule above.

As stated, after a subterm of the goal is h.o. unified with the LHS of a h.o. wave rule, the RHS must be $\beta_0\eta$-normalized (at the abstract \\. @ level) before being replaced in the goal. If the result is a degenerate wave front it is removed altogether.

The goal must first be put into its abstract syntax form before our routines can be applied and converted back after a successful ripple. As this only has to be done once per ripple, or even once per series of uninterrupted ripples, and is linear, the overhead is small.

The code for our routines can be found in appendix F. Higher-order wave rules are stored using the predicate `ho_wave/2`. As presently implemented the annotation contains no direction or type information for the wave front all fronts are assumed to be real and moving outwards. The top-level unification predicate is `unify/1`. This takes a list of equalities as its argument, and instantiates Prolog variables if unification is possible. The result is not normalized. Routines are included for various forms of normalization and pretty-printing of terms.

4.12 Conclusions

We have been able to successfully using *CLAM*'s existing method set and induction proof plan to prove both the synthesis and decidability theorems for several tactics. We required principled extensions to the rewriting facilities, viz. context-sensitive rewriting and higher-order wave rules. Using the notion of context we able to formulate criteria for elimination of quantifiers. With these extensions, and several smaller extensions, in place synthesis was successful.

We have seen how these proofs may themselves be used as tactics via the reflection mechanism. In the next chapter we shall see how they may be compiled to pseudo-tactics.

Chapter 5

Tactic compilation

5.1 Introduction

In this chapter we finally achieve the object of synthesizing tactics which satisfy Clam method specifications, albeit greatly simplified ones. We do this by looking at the extract from the decidability proof for a tactic (the second form where all "Prolog" variables in the `preconds` are universally quantified) and the synthesis proof. We use the former do derive a Prolog program which succeeds exactly when the preconditions are true. and the latter to calculate the output when this is the case. In addition to this principal aim, we are also interested in using the technique described as a general means of synthesizing logic programs formally; for this reason we shall try as far as possible not to rely on any special properties that the kinds of proof discussed in the previous chapter may possess.

This being said, the immediate requirement of the resulting logic programs is for use as pseudo-tactics. We can use them in the same manner as *CLAM*'s proof planner searches for a meta-level proof. If the programs are called, say, `pre(I,X,Y)` and `eff(I,X,O)`, then we have the pseudo-tactic

```
pseudo(I,X,Y,O):-pre(I,X,Y),eff(I,X,O).
```

one for each tactic. We use these as we would *CLAM* methods to build a plan. Once a successful one has been found (simulated) we use the reflection mechanism

to turn it into a proof as described in section 4.2. It may appear that this circular exercise is rather futile we have gone from an existing method, backed up by a tactic, to its formal specification, proved that it is satisfiable, and arrived back at a Prolog pseudo tactic. The difference is that we know that the latter is correct. As methods become more complicated and theorem provers more powerful this correctness will be as important here as for any programming project.

5.2 Compilation

Our idea is to derive a Prolog program `prog(Tr,Args)` from the OYSTER term `t`, where `Args` are all the free variables of `t`, such that the following holds: for all ground substitutions θ for the variables `Args`:

$$\vdash_{\mathrm{SLD}} \texttt{prog(inl(\_),}\mathit{Args}\theta\texttt{)} \Longleftrightarrow\!>> \texttt{isl(t}\theta\texttt{)} \tag{5.1}$$

where $\texttt{isl}(t) =^{\mathrm{def}} \texttt{decide}(t; u.\texttt{true}; u.\texttt{false})$ is the OYSTER predicate which asserts that `t` evaluates to a left injection. Here `t` must be the inhabitant of a specification type; in most cases we consider either the extract of a synthesis theorem or the inhabitant of dequantified versions of such a theorem. Where the extract comes from a decidability theorem we are interested in a simple yes/no answer. Where the extract comes from a $\forall\exists$ synthesis theorem we will in addition want to calculate any witnesses implied by the specification, e.g. the contents of the output slot. One obvious way to find such a program is simply to normalize `t` and check if the outermost constructor is `inl`. This, however, as a Prolog program will work only in the mode `prog(+,-)`. We wish to use Prolog's backtracking mechanism to search for values for `Args`. To do this `prog` searches for all possible values of `Args` which results in `t` being a left injection.

The primary notion is that of the decidability of a proposition, i.e. an instantiated predicate. For the present we consider only first-order predicates. Hence, the propositions we consider will all be in the universe of small types, `U1`.

Such a translation works for simple programs. However, we are working in a constructive logic where failure, as well as success, of a sub-program can pass back

information. For this reason we need to allow predicates to explicitly make use of such information. We do this by adding a *truth variable* to the predicate. Thus we now have a generalization of 5.1:

$$\vdash_{\mathrm{SLD}} \mathtt{prog}(Tr, \overline{Args}) \iff t(Args) = T \tag{5.2}$$

where Tr is a Prolog term corresponding to the OYSTER term T. In order to regain a standard logic program from this we must transform the resulting logic program for the case when `T` is unified with `inl(_)`.

The logic programs synthesized below are quite large and awkward. They are not as poor as they appear for the following reasons:

- extensive use is made of auxiliary functions. In most cases this is not necessary (as they are not recursive) and so could be moved in-line. Many Prolog compilers do this as a matter of course.

- where auxiliary predicates are used they appear to have an abundance of arguments. This is because they usually correspond to terms deep within the extract, at which points there may be many bound variables. Most of the arguments can be eliminated since they are either duplicated or not used.

- the top level type is that of a decidability theorem so, roughly speaking, we can ignore all text inside an `inl(_)` or an `inr(_)`. As stated above, this cannot in general be ignored, but in most cases can.

- these are constructive logic programs. Just as we get a standard logic program for `pred` by unifying T with `inl(_)`, we get a standard logic program for `not_pred` by unifying T with `inr(_)`.

There are (at least) two ways to improve the resulting logic programs. We can "optimize" the raw logic program after translation, e.g. unify T with `inl(_)`, remove trivial clauses, move auxiliary functions in-line, remove unused or duplicated arguments. At the expense of complicating the translation process a little, a lot

of this burden could be moved there: by looking ahead we could check which variables are free (i.e. used) in a term and only use these as parameters in auxiliary functions.

We first give as a motivating example a simple (non-recursive) program, `test`, to decide whether a given integer is 0 or 1. We have the straightforward logical definition:

$$\texttt{test}(x) \;\Leftrightarrow\; x = 0 \texttt{ in int} \vee x = 1 \texttt{ in int}$$

Transforming this into an executable logic program starts with its

$$\vdash \forall x : \texttt{int}.\delta(\texttt{test}(x))$$

This has extract:

```
\x.
  decide(int_eq(x;0;inl(axiom);inr(axiom));
    v2.inl(int_eq(x;0;inl(axiom);inr(axiom)));
    v3.decide(int_eq(x;1;inl(axiom);inr(axiom));
        v7.inl(int_eq(x;0;inl(axiom);inr(axiom)));
        v8.inr(\v10.axiom)
      )
  )
```

We can reason as follows. The value of body on line 3 is always going to be a left injection. Similarly line 5. The converse is true of line 6. Thus the body of line 4 is a left injection exactly when its `decide`'s first argument is. This argument in turn is a left injection exactly `x` is 1. In such a bottom-to-top fashion we are able to derive the corresponding logic program:

```
test_dec(A,B) :-
        (   test_dec_1(inl(C),B),
            A=inl(int_eq(B,0,inl(axiom),inr(axiom)))
        ;   (   test_dec_2(inl(D),B),
```

```
                A=inl(int_eq(B,0,inl(axiom),inr(axiom)))
            ;   A=inr(lambda(v10,axiom)),
                test_dec_2(inr(E),B)
            ),
            test_dec_1(inr(F),B)
        ).

test_dec_2(A,B) :-
        (   B=1,
            A=inl(axiom)
        ;   A=inr(axiom),
            not(B=1)
        ).

test_dec_1(A,B) :-
        (   B=0,
            A=inl(axiom)
        ;   A=inr(axiom),
            not(B=0)
        ).
```

After simple optimization described below the result is

```
prog(B):-B=0;not(B=0),B=1.
```

This is not a good Prolog program because of the ordering of the literals in the second disjunction. However, floundering caused by such ordering can be prevented by a simple reordering of literals. The use of `not` here does not imply a closed world assumption (CWA): since our predicates are SLD-decidable we can soundly use negation-as-failure. We expect the predicate inside to be fully ground by the time it is called and therefore decidable.

Having given an idea of what is going on in the previous section we now give a more involved example. This is the synthesis of the logic program for deciding

whether a integers is a member of a list of integers (i.e. the traditional `member/2` predicate) and involves simple recursion over lists. We define:

$$\texttt{member}(x, l) \;=\; \texttt{list\_ind}(l; \texttt{False}; h, t, v.x = h \texttt{ in int} \vee v)$$

and prove the theorem

$$\vdash \forall x : \texttt{int}.\forall l : \texttt{int list}.\delta(\texttt{member}(x, l))$$

with resulting extract:

```
\x.
  (\l.
    list_ind(l;
      \oy{inr}(spread(term_of(member1)(x);v2,v3.v2));
      h,t,v.
        decide(int_eq(x;h;inl(axiom);\oy{inr}(axiom));
          v6.inl(spread(term_of(member2)(x)(h)(t)(axiom);
                        v13,v14.v14(axiom)));
          v7.
            decide(v;
              a1.inl(spread(term_of(member3)(x)(h)(t)(\v21.axiom);
                            v22,v23.v23(a1)));
              b2.\oy{inr}(\v25.axiom)
            )
        )
    )
  )
```

and raw translation:

```
member_dec(A,B,C) :-
        member_dec_1(A,B,C).

member_dec_1_2(A,B,C) :-
```

```
        member_dec_1(A,B,C).

member_dec_1_1(A,B,C) :-
        (   C=B,
            A=inl(axiom)
        ;   A=inr(axiom),
            not(C=B)
        ).

member_dec_1(A,B,C) :-
        (   B=[],
            A=inr(spread(apply(theorem(member1),C),bind([v2,v3],var(v2))))
        ;   B=[D|E],
            (   member_dec_1_1(inl(F),D,C),
                A=inl(...)
            ;   (   member_dec_1_2(inl(G),E,C),
                    A=inl(...)
                ;   A=inr(lambda(v25,axiom)),
                    member_dec_1_2(inr(H),E,C)
                ),
                member_dec_1_1(inr(I),D,C)
            )
        ).
```

This can be optimized to:

```
member(B,C):-B=[D|E],(C=D;member(B,E),not(C=D)).
```

5.3 The translation

Translation is divided into two mutually recursive steps. The first takes removes all outermost lambda abstractions so that the job of translating the extract $\lambda x.t$ in $\forall x.\delta(P)$ into a logic program becomes the job of translating `t` in $\delta(P)$ into the logic program body for the head `pred(Tr,x)`.

The translation is as follows:

$pred(Tr, \overline{x})$	t
$Tr = \texttt{inl}(z)$	$\texttt{inl}(z)$
$Tr = \texttt{inr}(z)$	$\texttt{inr}(z)$
$a = b, l*; \texttt{not}(a = b), r*$	$\texttt{int\_eq}(a; b; l; r)$
$d * (\texttt{inl}(u)), l*; d * (\texttt{inr}(v)), r*$	$\texttt{decide}(d; u.l; v.r)$
$aux(l)$	$\texttt{list\_ind}(l; b; h, t, v.r)$

where aux(Tr,l,$\overline{x}$):-l=nil,b*;l=[h t],r*[aux(t)/v]

5.4 Soundness and Completeness

One of the advantages of using the existing NuPRL logic (of which OYSTER is a re-implementation) is its soundness. The implementational soundness of NuPRL is highly likely by virtue of its large user base. Its theoretical consistency is shown in the work of various members of the NuPRL group (Allen, 1986; Howe, 1989; Mendler, 1986).

We now show the soundness of the translation described above w.r.t. the semantics of NuPRL. The proof is informal since we need to mention objects both of NuPRL and a first-order logic programming language.

Before proceeding with the proof we need to define the notion of a first-order data type. Objects in NuPRL can belong to a very rich set of types, and cannot

all be meaningfully translated into a first-order language. In particular, a first-order language has no way of expressing arbitrary application or binding. For this reason, although we do not care how terms are used inside a proof, the ones that we must of necessity translate, i.e. the terms which will instantiate the top-level variables, must be representable in a first-order language. Thus we define a *first-order data type* (or simply *data type*) as a NuPRL type all of whose constructors belong to the set {`unary, #, |, pnat, int, atom, rec`}. Note that we do not allow function types in this definition and that all types are non-empty.

Our completeness result is a corollary to the following. We show that if an Oyster term t has data type β and all of its free variables have a data type as their type, then we can find a Prolog program with top predicate *pred* such that, w.r.t. the completion, and for all ground terms *Params* and o:

$$>> t = o \text{ in } T \Longleftrightarrow \vdash_{SLD} pred(Params, o) \qquad (5.3)$$

Here *Params* is exactly the free variables in t. Often we will not need to use all the free variables to define a predicate, either because they appear in parts of the extract which don't get used during logic program synthesis or because they disappear as subparts of t get evaluated. Free variables may also be implicitly present in induction hypotheses though not present in the extract itself; by keeping track of when this occurs, e.g. by remembering that, say, the induction hypothesis corresponding to variable `v2` contains the free variables `v0` and `v1`, it is not necessary to keep accumulating variables the deeper we go into an extract we showed an example of this in the `merge` extract. For these three reasons it is simpler to remove redundant parameters at the post-synthesis stage, by which time all dependencies are known. This process of defining a program corresponding to t can be seen as partially evaluating the Prolog query `? eval(t,O)`. Our objective is to give a "natural" partial evaluation corresponding to the logic used in synthesizing t. By restricting the free variables to have data types as types we ensure that we can do this in Prolog (first-order). We could, of course, have written a backtracking evaluator and used the query given above. However, our objective in using this technique to realise tactic synthesis is to produce a Prolog program independent

of a particular system; we also wish to show that this paradigm of logic program synthesis is generally applicable as in (Bundy *et al.* 1990a; Wiggins, 1992).

The proof is by induction on the evaluation order of t. A property of the OYSTER type theory is that all well-typed closed terms evaluate under normal-order reduction to a canonical term. Hence, our induction, whose base case is signalled by t's being a canonical term, is well-founded. NuPRL's canonical terms may contain non-canonical subparts which we don't want present in the Prolog output term; these are dealt with by further reductions which we explain below. [See footnote 3 on p. 100 of (Constable *et al.* 1986). This may suggest a slightly more efficient compilation technique.]

We evaluate t under normal-order reduction until a free variable, X say, becomes the outermost redex. In all cases except that where t *is* X, X will be, in NuPRL terminology, in an *argument place*, whose non-canonical constructor requires that that position be occupied by an appropriate canonical term before evaluation can proceed further. We perform a case-split based on these non-canonical constructors. In the case where these constructors are, additionally, recursive (e.g. *p_ind*, *list_ind*) we must make use of an assumptions corresponding to induction hypotheses. This form of proof, for a particular t, is formalizable in NuPRL. The proof itself is not wholly formalizable since it employs metatheoretic notions such as redex, principal argument.

We give the rules for the non-recursive non-canonical constructors first. The notation $C[.]$ is used to indicate that the redex of the term $C[.]$ is $[.]$. All formulas are to be understood as universally closed.

$$\frac{C[b[L,R/l,r]] = o \leftarrow pred1(L,R,Params,o)}{C[\mathtt{spread}(X;l,r.b)] = o \leftarrow pred(Params,o)}$$

where $pred(Params,o) \leftrightarrow X =< L,R > \wedge pred1(L,R,Params,o)$.

$$\frac{C[a[L/l]] = o \leftarrow pred1(L,Params,o) \qquad C[b[R/r]] = o \leftarrow pred2(R,Params,o)}{C[\mathtt{decide}(X;l.a;r.b)] = o \leftarrow pred(Params,o)}$$

where $pred(Params,o) \leftrightarrow (X = \mathtt{inl}(L) \wedge pred1(L,Params,o)) \vee (X = \mathtt{inr}(R) \wedge pred2(R,Params,o))$.

$$\frac{C[c] = o \mapsto pred1(Pars, o) \quad C[d] = o \mapsto pred2(Pars, o) \quad b = o \mapsto pred3(Pars, o)}{C[\texttt{pnat\_eq}(X; b; c; d)] = o \mapsto pred(Pars, o)}$$

where $pred(Pars, o) \mapsto pred3(Pars, o') \wedge ((X = o' \wedge pred1(Pars, o)) \vee (X \backslash = o' \wedge pred2(Pars, o)))$.

$$\frac{C[c] = o \mapsto pred1(Pars, o) \quad C[d] = o \mapsto pred2(Pars, o) \quad a = o \mapsto pred3(Pars, o)}{C[\texttt{pnat\_eq}(a; X; c; d)] = o \mapsto pred(Pars, o)}$$

where $pred(Pars, o) \mapsto pred3(Pars, o') \wedge ((X = o' \wedge pred1(Pars, o)) \vee (X \backslash = o' \wedge pred2(Pars, o)))$.

$$\frac{???}{C[X(a)] = o \mapsto pred(Params, o)}$$

Note: this situation can never occur since we restrict our free variables to have data type type and all rules preserve this property.

$$\frac{}{t = o \mapsto pred(Params, o)}$$

where t is a canonical term. We are done and our predicate is defined as

$$pred(Params, o) \mapsto t = o$$

t and all its subterms must be canonical since there are no redexes present in t.

We now gives the rules for the recursive non-canonical constructors.

$$\frac{C[b] = o \mapsto pred1(Params, o) \qquad C[s[P/p]] = o \mapsto pred2(P, Params, o)}{C[\texttt{p\_ind}(X; b; p, v.s)] = o \mapsto pred(Params, o)}$$

where the second premise is under the assumption that $C[v] = o \mapsto pred(\ldots, P, \ldots, o)$.

This rule presents two problems. First, it introduces a free variable v which is not necessarily first-order and thus may violate our restriction. Second, it is not clear that the assumption $C[v] = o$, although universally quantified over all variables apart from P, will always of the correct form to close of a proof.

The first problem is not real since we have in Prolog the advantage of arbitrary recursion: we replace all calls to v with calls to $pred(\ldots, T, \ldots, o)$. This way out

of the first problem presumes we have answered the second: that all calls to v will will be of the form $C[v]$ for some instantiation of the universally quantified variables therein. This is not always the case. We hope at a later date to provide a precise characterization of the class of proofs for which the translation process succeeds.

Similarly with:

$$\frac{C[b] = o \leftarrow pred1(Params, o) \qquad C[s[H, T/h, t]] = o \leftarrow pred2(H, T, Params, o)}{C[\texttt{list\_ind}(X; b; h, t, v, s)] = o \leftarrow pred(Params, o)}$$

where the second premise is under the assumption that $C[v] = o \leftarrow pred(\ldots, T, \ldots, o)$ and we have the definition $pred(Params, o) \leftrightarrow (X = [] \wedge pred1(Params, o)) \vee (X = [H|T] \wedge pred2(H, T, Params, o))$. Note that, barring the exception of the new possibly higher-order variable v as in Peano induction, both the new variables H and T have data types as types since X did.

Having shown how we translate auxiliary functions into Prolog programs we immediately have a means of logic program synthesis from decidability proofs. Since we are only interested at the top level in whether the inhabitant of the decidability assertion $d(P)$ is a left or right injection we can treat the assertion as a data type. [If one wished to be precise we can "squash" the top-level type with e.g. $decide(D; l.\texttt{inl}(axiom); r.\texttt{inr}(axiom))$ which is a member of the data type $(0 inp nat)|(0 inp nat)$ whenever D is a decidability assertion.] Thus, if t is the extract of our decidability proof, and if $pred(Params, O)$ is the corresponding logic program when t is viewed as an auxiliary function, then $pred(Params, \texttt{inl}(\_))$ is the program corresponding to P and $pred(Params, \texttt{inr}(\_))$ is the program corresponding to $\neg P$. We show below how using a delaying meta-interpreter for these programs allows us to use them multi-mode.

Extract reduction

Above, extracts were simply given. However, these are different from the immediate extracts of the decidability theorems. The extracts shown have had all definitions expanded and have then been reduced by OYSTER's `eval` function. `eval`

carries out eager reduction on all subterms. This is safe with all extracts we use since none of our theorem proving procedures will introduce non-terminating subterms. We prevent this reduction being applied to certain terms, however, where we wish to retain structures arising form the decidability proof. Since these structures are typically cut into the proof by use of a lemma they occur in extracts as `term_of` terms. Examples are non-primitive induction/recursion schemes such as that for `metaterm`, predicates for which we already have a decidability lemma and hence are able to compile a sub-program, and purposely introduced connectives which are compiled to particular logic programming connective. Examples of the latter are the extracts of the lemmas which distribute the δ's around propositional connectives. For example the lemma `d_and`

$$\vdash \forall a, b : \mathtt{U1}.\delta(a) \wedge \delta(b) \rightarrow \delta(a \rightarrow b)$$

has extract . We can either evaluate `term_of(d_and)` when it occurs or use a custom translation one translation may call its goals left-to-right, the other right-to-left. `d_and` behaves like a higher-order logic programming construct. In this way such things as modular logic programming and Prolog cliches could be incorporated into synthesized logic programs if the corresponding lemmas are used in the proof of the decidability theorem.

Another example where this is useful is in the use of data types where equality is decidable. For instance, instead of generating an auxiliary predicate which decides if two integer lists are equal, we leave `term_of(int_eq_dec)(l)(t)` unexpanded and translate it simply as Prolog unification, `l=t`.

Translation is still very primitive at present. For example, we only consider list recursion on variables, not arbitrary terms. In the integer equality case split we assume a and b are integers or integer variables. Again, these could be arbitrary terms.

5.5 Optimization

The optimizations that we carry out are divided into three types: those that are carried out before translation (i.e. on the OYSTER extract term), those carried out during translation, and those after translation (i.e. on the raw logic program).

5.5.1 Pre-translation optimization

As outlined above the optimizations available here consist of term normalization and choices over which `term_of` terms to expand.

5.5.2 Translation optimization

There are many optimizations which could be carried out during or after translation. After trying arity reduction as a post-translation optimization it is clear that some optimizations are best carried out at translation time. The single reason for this is the form of the information we have available: during translation, at any point, we know exactly which variables are bound, whether variables will be used in a sub-program, etc. Thus, information that would be distributed across the whole of the translated logic program is here highly localized. For this reason we feel it best to carry out all optimizations which make use of non-local information at translation time. This has been done for arity reduction, which now consists simply of checking for which variables are free (i.e. used) in a subterm and using only these as arguments.

5.5.3 Post-translation optimization

The optimizations in this class that we have implemented so far are:

- Arity reduction: this simply checks for singleton or repeated variables in the head of a clause and transforms them to a simplest form. The transform is

then extended program wide. This operation is carried out repeatedly until no more simplifications can be made.

- Injection elimination: a predicate of the form `pred_dec(TruthVar,...)` is split into two predicates `pred(...)` and `not_pred(...)`. Often, the original and `not_pred` will not be used by the program and so can be thrown away.

At present the translation process is very primitive. It expects most of the terms it is translating to belong to a decidability type (i.e. `inl`$(\cdot)$ or `inr`$(\cdot)$). In these simple examples this was always the case. However, it easy to imagine cases where one has, instead of a variable over which list recursion is taking place, another list recursion. Thus we need to allow for the synthesis of auxiliary predicates for calculating such nested recursion. This can be accomplished by adding another argument to the `translate` predicate for the type of the term being translated. At the top level this will always be $\delta(.)$, but in general we must provide translations for all primitive types considered, e.g. `int`, `int list`, `pnat`, etc.

All optimizations, although trivial, have so far been done by hand. These are easily automated.

5.6 Use of compiled pseudo-tactics

Atomic tactics that we have successfully synthesized can be translated as shown above. [We supply as appendices the compilation of the earlier `eval_def` synthesis and decidability theorems]. They may also be used directly with the reflection mechanism. There are some tactics, however, where a pseudo-tactic is the sole means by which it can be used. The reason for this is the limited nature of our reflection mechanism. First, since we lift only terms and not sequents, we are unable to express naturally the correctness of hypotheses without giving rise to a contradiction. This means we are unable to specify a tactic such as strong fertilization. Second, by choosing to use type theory we are restricted to synthesizing terminating tactics. There are some cases, for example a semi-decision procedure,

where we might be willing to forgo totality for completeness. In the latter case we can use Prolog's unconstrained search to build a plan from previously compiled pseudo-tactics but, of course, we lose the guarantee of termination.

An example of such a pseudo-tactic of direct importance to *CLAM* is `ripple`. A simple definition of this would be:

```
ripple:= wave
```

We are able to synthesize the terminating tactic `wave` as we did `eval_def` and compile this to a pseudo-tactic, say `wave_pseud(I,Pos,Exp,O)`. We could then write a pseudo-tactic for ripple:

```
ripple(I,[],I).
ripple(I,[(Pos,Exp)|T],O):-
    wave_pseud(I,Pos,Exp,Temp),
    ripple(Temp,T,O).
```

We suggest ways of synthesizing a pseudo-tactic for weak-fertilize in chapter 7.

Chapter 6

Related Work

6.1 Introduction

In this chapter we describe in some detail work related to that of the preceding chapters. The area of interest might be summed up as "verification and synthesis of proof procedures". The approaches taken can be divided into two broad camps. Howe has used the terms "syntactic reflection" and "semantic reflection" to differentiate between systems which internalize (part of) their own proof theory, usually via some kind of "reflection principle" in the form of an axiom or inference rule, and those which internalize only (part of) their semantics, i.e. a *meaning* mapping between terms and their values. Since the mechanism we describe in chapter 3 is the latter kind, the major part of this chapter describes two other "semantic" systems, those of Boyer and Moore, and Howe[1]. The descriptions below will pay particular attention to areas which we found difficult, e.g. how meta-theorems are used in extensions of the theories in which they were originally proved correct

[1] According to (Boyer & Strother Moore, 1981), Brown (Brown, 1977) appears to have been the first to use a meaning function in this manner, though not "in a way that permits its mechanical application."

(monotonicity), implementing reflection mechanisms efficiently, formalizing extra-logical concepts (in our case, *CLAM*'s database), and applying reflection to whole goals rather than just terms within them.

For comparison with the work in the thesis the following aspects have been described in detail:

1. The objective(s) which the reflection mechanism is intended to meet. Give a historical perspective: when was the system devised; was it a first; has it been overtaken since then?

2. The object logic, its strength and expressiveness. E.g. does it have (and does the reflection mechanism make use of) a notion of evaluation/computation?

3. The meta level: is this part of the object-level; how is the object logic represented; is it sound and consistent; is it a conservative extension of the object-logic; the space-complexity and naturalness of this; how large a fragment of the object logic can be represented and what are the ramifications of any restrictions, for example...

4. What kind of knowledge can be stored in the form of meta-theorems, e.g. is there something akin to Knoblock's *analytic preconditions*? Is this essentially different from what is possible without the reflection mechanism.

5. Is there a reflection theorem? Is it formal or informal; object- or meta-level?

6. Is there more than one meta level? What is the purpose of further levels and what is their *architecture*?

7. What facilities are provided for the user when formulating meta-theories? Must one first define useful syntactic operations, or does a good supply come with the mechanism? Can one direct a theorem prover (if applicable) to help in the proofs.

8. Is there any attempt at the automation of the proofs of meta-theorems? How far does this attempt go (whole proofs? lemmas?) and how successful is it?

9. Once proved, can meta-theorems be used by the theorem prover, so that a "bootstrapping" process is going on?

10. The transition between the object and meta levels. How are object-level formulas lifted; how are they lowered? What are the computational complexities of these processes, e.g. are a large number of well-formedness subgoals produced? Also:

11. If the meta-theorem requires computation (e.g. a decision procedure in the form of an analytic pre-condition), how is this carried out? Answer the same questions for this process as in the previous item.

12. How is monotonicity handled? I.e. if we extend the object logic (with, say, some new definitions), is the corresponding extension of the meta-level painless? Is there any such facility at all?

13. How can the meta-theorems be used? Is there a time/space/other advantage to their use in proofs? Is there a penalty; a trade-off?

14. Is there a (LCF-like) tactic language as such so that one can easily piece together meta-theorems to form new ones, for example? Is there a "nice" way of representing and using failure. Could such a facility be easily implemented?

15. Have any big examples been done with the reflection mechanism? Was there something essential about its use in these?

16. From the user's perspective: what level of sophistication is required to use the reflection mechanism? Does the user have to write tactics, prove meta-theorems, incorporate these in any way so that the theorem proving environment can make use of them?

17. Are there likely to be any problems in scaling the mechanism up for everyday (industrial strength) usage?

18. (Related to (1) and (4)) Do the specifications used in pre-conditions etc. involve a meta-level *system*. For example in $CLAM$ we not only talk about the syntax of object-level formulas, but also manipulate meta-level annotations to that (wave annotations) and refer to a meta-level database (wave-rules, reduction rules etc.). This presents us with a problem, for while the object-level system is monotonic (when a theorem is proved it stays proved) the meta-level system may not be (e.g. a proof which Clam finds may not be found when more rules are added since the search is not necessarily monotonic). Should monotonicity be a pre-requisite of any search procedure?

6.2 Boyer and Moore

We describe here the theory and use of metafunctions given in (Boyer & Strother Moore, 1
Knowledge of the logic and theorem prover NQTHM (Boyer & Moore, 1979) is assumed.

(Boyer & Strother Moore, 1981) covers five broad areas. First, the object-level aspects of the reflection mechanism and the obligations of correctness proofs are described. The validity of reflection is then proved in the central metatheorem. The largest part of the paper describes how the reflection mechanism is made extremely efficient by taking advantage of certain properties of NQTHM's implementation. A short section describes how NQTHM was used to prove the correctness theorem for the cancellation example (i.e. to verify the metafunction) and gives an idea of the likely difficulty of this problem in general. Finally, it is explained how metafunctions, once proved correct, can be used by the theorem prover. Running through the paper is the concrete example provided by the cancellation metafunction, `CANCEL`, for cancelling `PLUS` terms across an equality.

This work is the earliest we have found which implements semantic reflection (though see footnote on page 126); it is also the most efficient. Of most interest to us are the nature of the validity proof and the efficiency aspects of the im-

plementation. The latter demonstrates that it is possible to be both formal and efficient.

6.2.1 Aims

The aim of Boyer and Moore with this work is to allow the user to extend the theorem proving capability of NQTHM in ways not previously possible. Hitherto, the theorem prover has been a "black box" whose actions could only be influenced by the addition of lemmas for use in prescribed ways. There is no provision for the user to add tactics, in the LCF sense, to the system. The approach described in (Boyer & Strother Moore, 1981) allows the user to add executable code in the form of *metafunctions*, through which the formulation of "schematic lemmas" becomes possible `CANCEL` described below is a good example. In (Boyer & Strother Moore, 1981) it is proved in the metatheory of NQTHM that provably correct metafunctions may be soundly added to the system. Boyer and Moore also place a premium on efficiency here, the intention being that metafunctions be used as an everyday part of NQTHM. Indeed, a slightly optimized version of `CANCEL` is now incorporated in NQTHM as standard and has been used for large-scale problems.

The main obstacle to this overall aim is seen to be the obligation to prove metafunctions correct. These proofs are complicated enough to make them feasible only with the aid of a theorem prover, and this paper shows that NQTHM, with its particular ability in the domain of inductive proofs, can be a useful tool to this end[2]. The justification for continuing to use a reflection mechanism in the face of this problem are the same as those given in chapter 1, viz. that it is the best long-term hope for raising the level of reasoning in a system.

---

[2]Howe shows (see section 6.3) that it is possible, using powerful techniques and autotactics, to undertake such proofs without the aid of an inductive theorem prover though extremely tedious.

Although other metafunctions have been developed by Boyer and Moore (see (Boyer & Moore, 1979) for a decision procedure for propositional calculus), the emphasis in (Boyer & Strother Moore, 1981) is on one-off correctness proofs. There is no indication of the methodology for combining metafunctions which one might expect if metafunctions are to be used as the tactics of NQTHM.

6.2.2 Technical details

NQTHM (Boyer & Moore, 1979) is a mechanised, classical, quantifier-free, unsorted, first-order predicate logic with equality and is implemented in a dialect of LISP known as INTERLISP. (The absence of destructive operations from this subset of LISP is further evidence for our supposition in chapter 2 that, even for very efficient implementation of tactic-like programs, imperative languages are a poor choice. It also suggests a relatively spartan language such as type theory is sufficient.) It allows induction upto ε_0, and has facilities for the introduction of simple inductive "types" called *shells*, and provably total functions. NQTHM has a notion of computation called reduction; this always terminates. There is also a capability to add arbitrary axioms, though this is rarely used. A theory to which no arbitrary axioms have been added is called by Boyer and Moore *constructive*. NQTHM contains a powerful heuristic theorem prover capable of proving complex inductive theorems[3].

Expressions in NQTHM are written in a Church-style prefix notation, like that used in LISP. (To reduce the confusion NQTHM terms are written using upper case, and LISP terms, where possible, using lower.) Since these expressions only exist as internal representations, (Boyer & Strother Moore, 1981) uses the following convention to relate representation to NQTHM output. The LISP function s (for solidify), given a representing LISP term, produces another LISP term which *displays* as the corresponding NQTHM expression. The notation $|x|$ is used to

[3] *CLAM* includes a proof-plan-based reconstruction of these heuristics (Bundy *et al*, 1989b).

indicate the actual output produced by displaying the LISP object x. So the suitably well-formed LISP term *obj* represents the NQTHM expression |(s *obj*)|.

The original logic of NQTHM as presented in (Boyer & Moore, 1979) has undergone "certain minor revisions ... to undertake the meta approach conveniently." These are in the main related to the efficient and consistent representation of quotations, and a new class of lemma, `META`, is defined. In addition, some unconventional forms of axiom and definition are used. All are described below. Note, however, that it was not found necessary to make any changes to the theorem prover itself, its heuristics etc., to verify the metafunctions.

Boyer and Moore use a form of semantic reflection, similar to the mechanism described in chapter 3. Quotation of NQTHM terms is formalized in NQTHM itself. A predicate, `FORMP`, characterizes those terms which *correspond* to, i.e. which are quotations of, other terms. This is on a par with our use of the OYSTER type `metaterm`. The function `MEANING`, along with an assignment for giving a value to variables, is used to evaluate or "drop" quotations. This is equivalent to our OYSTER function `eval`. Also like our system, assignments take the form of association lists, the *standard a-list* being `(LIST (CONS "A" A) (CONS "B" B))` etc. and containing look-ups for all the variables in the lifted formula. `MEANING` takes the (arbitrary) default value `(TRUE)` when applied to a non-`FORMP` term. The steps for adding a new procedure are similar to ours: conceive some suitable term-to-term transformation; express this as an NQTHM function operating on the quotation of terms; use the theorem prover to prove the correctness theorem for this function; incorporate the metafunction into the system. We differ in that *CLAM* is used to synthesize the metafunction from user-provided specifications, and in the flexibility of use afterwards. If all these steps are successful NQTHM adds the new metafunction to the list of things it tries during the simplification process.

The *correctness theorem* for a function `fn` has the form:

```
(IMPLIES (FORMP X)
         (AND (EQUAL (MEANING X A)
```

```
                (MEANING (fn X) A))
        (FORMP (fn X))))
```

i.e. that for all input FORMPs X, one has to show the metafunction fn preserves MEANING under all assignments A, and that it produces a well-formed quotation term as output.

The quotation mechanism, what Boyer and Moore call the "correspondence between terms", can be confusing. First, the LITATOM shell, with constructor PACK, is used to quote symbols. A 0-terminated list of numerals corresponding to the ASCII codes of a symbol's name is the quotation of that symbol. E.g.

```
(PACK (CONS 78 (CONS 73 (CONS 76 0))))
```

is the quotation of the symbol NIL. The former is abbreviated to "NIL". (Note that 1 is an abbreviation for (ADD1 (ZERO)) etc.) A further abbreviation is employed so that (CONS A (CONS B "NIL")) may be written as (LIST A B) etc. The quotation of NQTHM terms is as follows:

- Variable symbols are quoted as described above, e.g. X as "X".
- The quotation of a function application is a list consisting of the quotation of the function symbol followed by the quotations of its arguments (if any). E.g. ZERO has quotation (LIST "ZERO")[4], and (NOT P) (LIST "NOT" "P").

The confusion arises because a second form of quotation is permitted for *explicit value terms* or EVTs (roughly: fully evaluated, ground terms). If t is such a term, (LIST "QUOTE" t) is also a quotation of t. To avoid ambiguity the use of QUOTE (and NIL) as a function symbol is forbidden in NQTHM. This second form of quote is a result of the implementation-level representation of terms and is explained below.

---

[4]Variables and nullary constants are distinguishable since the first are LITATOMs, the second LISTPs.

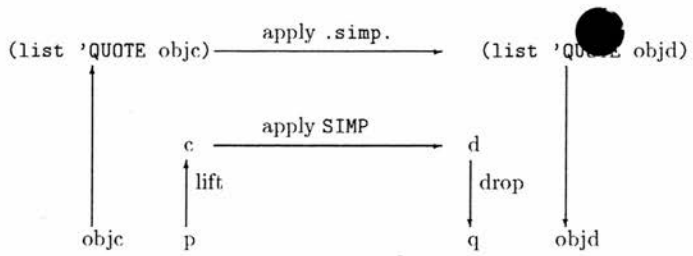

Figure 6–1: Boyer and Moore's reflection mechanism

The second form of quotation is only possible because what appear to be formulas in NQTHM are in fact terms. What appears at the top level as the formula `FORM` is actually the formula `FORM` $\neq$ `(FALSE)`, `FORM` being a term. What appear to be predicates are actually characteristic functions. Thus, although (like ours) Boyer and Moore's reflection mechanism is used only on the term level, it can effectively be applied to what appears to be the whole logic (unlike ours). Compare this with Howe's reflection mechanism (section 6.3) where actual propositions are lifted. Such mimicry of propositions by terms would be difficult in an intuitionistic logic since no finite truth-value model exists. For example, Howe's partial booleans (section 6.3) require the use of a third, proposition-valued injection when a simple true/false value cannot be obtained.

The reflection mechanism is used in much the same way as that of chapter 3. To rewrite a term `p` using the metafunction `fn`, NQTHM finds the quotation `c` of `p` and applies `fn` to it to produce a term `d`. Having proved `(FORMP c)`, the correctness theorem tells us that `d` is the quotation of some term `q`, say, and that `(EQUAL (MEANING c A) (MEANING d A))`. Substituting the standard assignment for `A` and evaluating further, we get `(EQUAL p q)` and may thus rewrite the goal to `q`. This process is illustrated by the inner loop of fig. 6 1. Compare this with our fig. 3 1. In practice this process is carried out at the implementation (LISP) level, illustrated by the outer loop of fig. 6 1. Boyer and Moore have utilised a feature of NQTHM's implementation which means that the lifting and dropping steps are done in constant time and space and that the evaluation step is as efficient as hand-coding the metafunction as an LISP routine.

With any reflection mechanism the problem of monotonicity arises: how can a

metafunction be used outside the theory in which it was proved correct. Boyer and Moore solve this in an interesting way, but one that is dependent upon special features of NQTHM. One requires a metafunction to be valid in any ordinary extension of the theory in which it was proved. This is achieved by separating the axiomatizations and definitions of FORMP and MEANING and their auxiliary functions called *meta-axioms* and *meta-definitions* and denying the user access to the latter. The meta-axioms, and therefore the validity of any correctness proof employing them, change monotonically as the theory is extended, as shown formally in the *metatheorem* (see below), whereas the meta-definitions do not. However, the meta-definitions are such that the meta-axioms are always satisfied. The reasons definitions are required at all are: superficially to satisfy the conventions of the NQTHM logic, and for evaluation at reflect-time where default values may be needed. It is only with respect to these default values that the meta-definitions are underspecificied by the meta-axioms. In fact, meta-definitions behave as if they are introduced immediately before reflection and removed immediately after. There is no means in OYSTER to prevent the use of definitions and we, following Howe, have made monotonicity explicit. Note, though, that where we have needed to simulate CL^AM predicates at the object-level, and these are theory dependent (e.g. `defeqn/2`), we too have found it necessary to subvert the idea of a fixed definition (see section 3.5).

For example, consider the meta-definitions and meta-axioms for the auxiliary functions APPLY and ARITY, the only ones which actually change with the theory, when PLUS is the last function defined in a theory. The meta-definitions are:

```
(APPLY X L) =
...
(IF (EQUAL X "PLUS" )
    (PLUS (CAR L) (CADR L))
    (TRUE))...

(ARITY X) =
...
```

```
(IF (EQUAL X "PLUS")
    2
    "NIL")...
```

and the meta-axioms:

```
...
(EQUAL (APPLY "PLUS" X)
       (PLUS (CAR X) (CADR X)))
...
(EQUAL (ARITY "PLUS") 2)
```

APPLY defines the meta-level PLUS in terms of the object-level one, and ARITY is used by FORMP. If in this state the function FOO, say, is undefined then at reflect-time (ARITY "FOO") will evaluate via the meta-definitions to "NIL" but one will not be able to prove (EQUAL (ARITY "FOO") "NIL" from the meta-axioms. The other reflection-related functions are similar. After defining a function FOO of arity 1 the meta-definitions are different:

```
(APPLY X L) =
...
(IF (EQUAL X "PLUS" )
    (PLUS (CAR L) (CADR L))
(IF (EQUAL X "FOO")
    (FOO (CAR L))
    (TRUE)))...

(ARITY X) =
...
(IF (EQUAL X "PLUS")
    2
(IF (EQUAL X "FOO")
    1
```

```
"NIL")...
```

but the meta-axioms are simply augmented by the following two:

```
(EQUAL (APPLY "FOO" X)
       (FOO (CAR X)))

(EQUAL (ARITY "FOO") 1)
```

Analogous meta-axioms are tacitly added every time a new function symbol is defined. Thus, the meta-axioms describe the language and meaning of the current theory for use in correctness proofs.

On consideration, FORMP and MEANING could not have normal definitions since they must change as function definitions are added to a theory. From (Boyer & Strother Moore, it is clear that the functions involved in the reflection mechanism, FORMP, MEANING, etc., are not themselves reflected in the way just described. Hence, although we said above that the mechanism appeared able to refer to the whole logic, it cannot reason about itself, and its "hard-wired" architecture is restricted to this one object-meta bridge. It is not clear why this limit has been imposed, apart from the resulting large jump in the complexity of the admissibility proofs given in (Boyer & Strother Moore, 1981) were it to be removed.

We now describe the metatheorem. For conciseness, if T is a theory, then define $MA[T]$ as T with the meta-axioms added as actual axioms, and $MD[T]$ as T with the meta-definitions added as actual definitions. Let T_1 be an constructive, ordinary[5] theory, and T_2 an ordinary extension of it. Further, suppose SIMP is an explicit value-preserving (see below), unary function defined in T_1. The main result that Boyer and Moore prove, what they call the *metatheorem*, is the following:

Suppose we can prove the correctness theorem for SIMP in $MA[T_1]$, that p is a term of T_2, c is a quotation of p w.r.t. T_2, and d is the reduction of (SIMP c) in

[5] An *ordinary* theory is one where none of the non-meta-axioms mentions the special functions FORMP, MEANING, etc.

T_2. Then: the reduction of (FORMP d) in T_2 is (TRUE), d is the quotation of some term q of T_2, and in T_2 we can prove (EQUAL p q).

Note the special attention paid to monotonicity ($T_1 \subseteq T_2$) in this theorem. Since the use of meta-axioms and meta-definitions is itself unorthodox in logic, the metatheorem takes the form of an admissibility result, i.e. if we ignore all the reflective machinery then we have in effect a conservative extension of T_2.

Although the metatheorem would seem to relate only to the object-level features of the mechanism, it is critically involved in the correctness of the implementation-level operation, via the hypothesis that SIMP is EV-preserving and the conclusion that (FORMP d) *reduces* to (TRUE). This connection is even clearer in the requirement that T_1 be constructive and therefore, by assumption, consistent. For were one to assume the reduction property when it was not in fact the case, NQTHM's implicit use of lemma 18 (see later) and implementation are such that totally unpredictable behaviour, including a system crash, may result. In normal use of NQTHM the ordinariness and constructiveness requirements on T_1 and T_2 will always be satisfied.

On a more pragmatic note, it is not clear how NQTHM ensures that a META lemma and its associated metafunction are used only in theories which are at least as strong as those in which they were defined. It is easy to show that use in a weaker theory results in inconsistency. Presumably along with each META lemma are recorded the definitions in force at the time it is proved, and NQTHM ensures the same are present before it may be used for reflection.

The requirement for a metatheorem shows up the biggest difference between Boyer and Moore's mechanism and that of Howe's and ours: the absence of an explicit environment in the former. What we do at the object level is done in (Boyer & Strother Moore, 1981) by the use of meta-axioms etc. We have already noted some of the advantages simplicity (since monotonicity problems are taken care of by a metatheorem), ease of use, and, to some extent, efficiency of using an implicit environment in the reflection mechanism, and some of the disadvantages non-portability, opacity and inflexibility. However, one of the major disadvantages is mentioned by Howe (Howe, 1988a). Having an explicit environment allows

one to abstract it so that one may, for example, write general metafunctions to work on any commutative monoid and then instantiate these as the need arises (Howe, 1988a). It would appear impossible to provide such abstraction in Boyer and Moore's mechanism. While abstractness is not an important consideration for the simple examples to which reflection mechanisms have so far been applied, it will become so as larger scale work is attempted.

The object-level explanation above of the reflection mechanism is quite similar to ours. But it is in the internals of Boyer and Moore's mechanism that some of the most novel features lie and where we most differ.

The form the quotation mechanism takes is driven by the internal representation of NQTHM terms. The objective is to make the lifting and lowering of NQTHM terms very efficient. Boyer and Moore achieve this very simply: if the LISP term *obj* represents the NQTHM term t, then (`list 'QUOTE` *obj*) represents *a* quotation of t (Boyer & Strother Moore, 1981, lemma 19). To enable an equally simple dequotation process a restriction is placed on representations of the form (`list 'QUOTE` *obj*): they must represent an explicit value term (explicit quotations are always of this form). This allows the system to assume that the NQTHM term represented need not be evaluated before being dequoted by stripping away the embedding `QUOTE` (Boyer & Strother Moore, 1981, lemma 18). Note that not all LISP terms of the form (`list 'QUOTE` *obj*) represent quotations. That class is decided by the NQTHM function `FORMP`.

A more general form of representation is also needed for those terms which are not EVTs. The following is used: represent an NQTHM variable by the literal atom formed from its name; represent the function application $(f\ a_1 \ldots a_n)$ by the term (`list` $F\ b_1 \ldots b_n$) where F is the literal atom formed from the name of f and b_i represents a_i.

The `QUOTE` form of representation allows a very compact representation for the common NQTHM shell types. `LITATOM`s are represented by LISP literal atoms[6].

[6]This also allows internal LISP techniques such as indexing and hashing to be used.

LISTPs by LISP conses, and NUMBERPs by LISP integers. These are the types used in NQTHM quotations (see below) and this compact representation is an integral part of the simple lifting and lowering process mentioned above. For example, the NQTHM term (NOT P) is represented by (list 'NOT 'P). Its quotation, the NQTHM term (CONS "NOT" (CONS "P" "NIL")) is represented by (list 'QUOTE (list 'NOT 'P)).

If the simple lifting procedure above is to apply uniformly, then the LISP term (list 'QUOTE (list 'QUOTE obj)) must represent the quotation of the term represented by (list 'QUOTE obj). To accommodate this Boyer and Moore have defined two forms of quotation (described previously) analogous to the two forms of representation (this explains why the phrase "*a* quotation of ..." was used above instead of "*the* quotation of ...").

Boyer and Moore are not clear about the non-determinism of representation, i.e. that some terms may have more than one representation (e.g. (list 'QUOTE 0) and (list 'ZERO) both represent the NQTHM term (ZERO)). It appears that the QUOTE form was intended originally only for lifted terms but later adopted for all EVTs for efficiency reasons. It also appears that lifting is always carried out by simply embedding the representation in a QUOTE and that, therefore, terms represented in the QUOTE (resp. general) form become quotations of the "QUOTE" (resp. general) form. The induced distinction between the two forms of quotation is critical to how metafunctions behave: once a QUOTE form is reached, i.e. an NQTHM term of the form (LIST "QUOTE" t), the metafunction no longer has access to the intensional structure of the term t, merely its denotation as an EVT. One could argue that one may apply *theory* rather than metatheory when manipulating EVTs, but this misses two points. First, that one can no longer write metafunctions in a uniform manner but must allow for the presence of either type of quotation, or at the very least must treat differently terms which happen to be EVTs. Second, the addition of shells to a theory changes the definition of EVT so that the *current* theory is no substitute for metatheory. These points aren't discussed in (Boyer & Strother Moore, 1981) since CANCEL does not need to manipulate EVTs, but the same certainly cannot be said of all useful metafunctions.

The final piece in the arrangement for efficient implementation of reflection deals with the execution of the metafunction. As with lifting and lowering this is performed at the implementational rather than object level. Boyer and Moore define a class of functions in NQTHM called *explicit value-preserving*. These are defined w.r.t. the intension of the function, and as far as we can see encompass all functions introduced using the definition principle though this is never stated. One of the properties (though not the definition) of EV-preserving functions is that an EVT is returned as result when EVTs are given as arguments. Every time an EV-preserving function `FN` is defined in NQTHM a compiled version called `.fn.`, a LISP routine operating directly on representations, is also stored and has the following property: if $c_1, \ldots, c_n$ are EVTs represented by `(list 'QUOTE` obj_1`),...,(list 'QUOTE` obj_n`)`, then `(list 'QUOTE (.fn.` $obj_1 \ \ldots obj_n$`))` represents the reduction of `(fn` $c_1 \ \ldots c_n$`)`. This property and the existence of such compiled functions can be shown by induction over the NQTHM histories (Boyer & Strother Moore, 1981, section 8). Since compiled functions are themselves composed only from earlier compiled functions there is no unpacking/repacking penalty involved in working on this compact representation of EVTs. During compilation several optimizations are also applied, including in-line expansion, redundancy elimination, and common sub-expression and tail-recursion optimization. Since quotations are always EVTs, EVTs are always represented using the `QUOTE` form, and metafunctions are EV-preserving, the above property results in extremely efficient execution of metafunctions approximately the same as LISP functions. We suggest how similar compilation could be achieved in type theoretic systems in section 7.2.2 as well as the more specialised case of chapter 5.

In summary we have the following, described by the outer loop of fig. 6 1. Lemma 19 establishes that NQTHM terms may be instantly lifted by embedding the representation in a `QUOTE`. Section 8 establishes that running the compiled metafunction produces the same result as the object-level one. Lemma 18 establishes that, under a certain assumption, terms may be instantly lowered by removing the embedding `QUOTE`; the assumption is provided by the metatheorem together with the correctness proof.

The assumption mentioned above is that (FORMP *d*) *reduce* to (TRUE). This is distinct from simple provability and is needed since lemma 18, and the lemmas leading up to it, are proved by induction on reduction. This style of proof has the twin advantages over induction on NQTHM proofs that: the induction is known to be valid since each function admitted to NQTHM is provably total; the textual form of the function definition concerned provides a guide to the shape of the proof.

There are two almost completely separate aspects to the reflection mechanism. There is the object-level justification given above, involving correctness proofs and use of FORMP and MEANING; and there is the implementation, constructed for maximum efficiency. The first, under the assumption of the consistency of NQTHM, guarantees that the second can never go wrong. The only points at which they meet is lemmas 18 and 19 and the justification of compiled functions. The metatheorem does not mention implementation though expressed in a form useful to lemma 18. This separation is quite different from Howe's and our own mechanisms where all work and justification is done at the object level including lifting, lowering and the running of metafunctions. Boyer and Moore eschew this approach believing, "were we to implement the mechanical application of of 'metafunctions' along the lines just described, the implementation would sink into a swamp of PLUS-trees." This has to some extent been borne out by the comparative inefficiency of the latter mechanisms. We offer suggestions for overcoming this problem in section 7.2.2.

There is a further example of how the object-level explanation of the mechanism is not used in practice: the meta-definitions of MEANING and FORMP are never used, i.e. neither is ever run as a function, only their meta-axioms used during correctness proofs. Reflection in practice follows the outer loop of fig. 6 1 and goes as follows. Lemma 19 justifies producing a quotation by embedding a representation *objc* within a QUOTE, i.e. it ensures that (FORMP |(s (list 'QUOTE *objc*))|) without running FORMP; the compiled version .fn. of the metafunction fn is applied to *objc* to produce *objd* (justified by the explanation of how compilation of EV-preserving functions works in (Boyer & Strother Moore, 1981, section 8), i.e.

that |(s (list 'QUOTE (.fn. *objc*)))| is (fn |(s (list 'QUOTE *objc*))|), which is guaranteed by the metatheorem to have the desired properties, viz. that MEANING is preserved and that (FORMP |(s (list 'QUOTE *objd*))|), where *objd* is (.fn. *objc*); instead of running MEANING with the standard a-list, dequotation consists of removing the QUOTE from (list 'QUOTE *objd*) leaving *objd*, justified by lemma 18 and the properties provided by the metatheorem. The only time an a-list is mentioned in the metatheorem is when the standard one is used to instantiate a correctness theorem. Thus, by using the metatheorem, instantiated with a correctness proof, and lemmas 18 and 19, the whole operation may be carried out in LISP — there is no need to run anything at the object-level. In fact, even QUOTE need not be used if one keeps track of when one is dealing with quotations.

Effectively the meta-definitions are never used and don't exist, being merely a pedagogical device for explaining the object-level behaviour of the mechanism. In essence, (Boyer & Strother Moore, 1981) shows that the changes to NQTHM which enable correctness theorems to be used as META lemmas constitute a conservative extension.

The writer of metafunctions for NQTHM has no specific tools to call upon but, as is shown with CANCEL, the standard corpus of number, list and bag theory provides a useful starting point. CANCEL does more than simply implement the cancellation law for PLUS as a lemma-based rewrite might; it cancels addends occurring arbitrarily deeply on both sides of an equation. For example, applying cancel to $B + (C + (I + X)) = (A + (I + J)) + (K + X)$ would produce $B + C = A + (J + K)$. CANCEL works by calculating the bag difference of the fringes of the two PLUS-trees and reforming this into a PLUS-tree as output.

Boyer and Moore say that proving the correctness of CANCEL was "not particularly difficult." The same cannot be said for the difficulty we encountered (chapter 4). There are several reasons for this disparity. The most obvious is the actual task: verification of a hand-coded function in (Boyer & Strother Moore, 1981) versus synthesis from specifications. (The absence of existential quantification from NQTHM precludes even expressing synthesis goals in the usual manner.) CANCEL, too, does not need to refer to meta-level databases or annotation as our

tactics do. The fact that the logic of NQTHM is classical and quantifier-free, while constraining the specification language available, makes proof easier. And while NQTHM, like OYSTER, constrains functions to be total, the lack of quantification means that questions of decidability, and the necessity of proofs thereof, never arise. We also attempt whole synthesis proofs, rather than deciding beforehand how they are to be broken into manageable lemmas. The above notwithstanding, `CANCEL` is of the same order of complexity as the tactics we wish to synthesize.

More detailed analogies can be drawn between our proofs and those in (Boyer & Strother Moo
The `FORMP` part of a correctness proof is roughly equivalent to well-typedness in ours, i.e. to showing that a function is a member of `metaterm` → `?metaterm`. A small difference is the theory-dependence of `FORMP`, for example its use of the arities of function symbols; we regard all metaterms as well-formed and simply map nonsense terms to a default value. In both cases the proofs involved are trivial. Boyer and Moore comment that, having divided the `FORMP` proof into four simple well-typing lemmas, no further user assistance was required; proofs involving induction over metaterms, bags and lists were completed automatically; and that "besides induction, the proofs required a good deal of simplification and the careful expansion of certain function definitions at the right moment."

The core of the correctness proof is the `MEANING` part. This is equivalent to the part of the synthesis goal stating the requirement that $i \sim_o o$. A standard corpus of list, bag and number theory is proved first, the latter including the object-level cancellation result on which the metafunction is based. Three relatively deep lemmas are then required before the main goal is proved automatically. In *CLAM* terminology they comprise a wave rule, a monotonicity rule and a reduction rule (see section A.3). Although several "eureka" notions are present in these, all appear within the definition of the `CANCEL` function. The top-level proof, again in *CLAM* terms, consists of rippling `MEANING` inward. Along with the forms of the three lemmas mentioned this suggests work on lemma speculation (Ireland & Bundy, 1992) may prove useful in this area of synthesis as well as more generally. There is no proof obligation equivalent to the remainder of our synthesis goal (the method specification) since the metafunction is provided by the NQTHM user.

The typelessness of NQTHM has both advantages and disadvantages. The advantages result from the simplicity of well-formedness: `FORMP` is much simpler than Howe's `wf` predicate, and there is no need for the complicated notion of type matching or the meta-type mechanism that goes with it. The disadvantages occur when dealing with the `MEANING` part of a proof, where NQTHM's type coercion, e.g. using the `FIX` function, must be accounted for both in formulating the metafunction and proving it correct. Fortunately, type coercion is a common problem in NQTHM and the standard heuristic eliminates the vast majority of cases.

Boyer and Moore state that no modification to the theorem prover was required to prove the correctness theorem, and this accords largely with our own experience. While we required specific extensions such as context-sensitive rewriting and higher-order wave rules (see chapter 4) to deal with the increased logical complexity of the propositions being proved, the overall heuristics provided by rippling and the other methods were found to suffice. Minor adjustments were required to guide synthesis.

Despite all the preceding detail about its properties the reflection mechanism in NQTHM is used in only one way. Once a metafunction's correctness theorem has been proved the result is stored as a lemma of sort `META`. From this point on the (compiled) metafunction is executed during the simplification stage of NQTHM's rewriting process; whenever the returned term is different from the input term it replaces it, using the metatheorem as justification. Note that metafunctions are total and that there is no failure mechanism as such. This simple pass/fail behaviour, as well as the absence of higher-order features from the language, does not allow metafunctions to be combined with tactical-like operators as is possible, for instance, in Howe's and our mechanisms — one could envisage defining new metafunctions in terms of others and using their correctness theorems in proving the new one, though this is never mentioned in ??. However, there is some heuristic control available to the metafunction writer. To help avoid fruitless execution the user may indicate that a metafunction should only be applied to a term whose outermost function symbol comes from a specified list. This is the same form of

control as used for "ordinary" lemmas. In the case of the `CANCEL` example, only on terms whose outermost functions symbol is `EQUAL` is rewriting attempted.

The advantages of a closely integrated reflection mechanism are balanced by its resulting non-portability. Besides some aspects of NQTHM itself which are state-dependent (e.g. the `COUNT` function), many more are added by the use of so-called meta-definitions. In conventional, monotonic logics, once a function is defined it is fixed for all time. The other likely portability problem is the efficient representation used. While any usable programming language will have equivalents of the basic data types used, lifting/lowering is unlikely to be a constant time/space operation. And the equivalent of compilation to `.fn.` form will be more complicated for a higher-order language — compare this to our chapter 5 for example. Finally, there is the burden of proving the equivalents of lemmas 18 and 19 and the metatheorem for a system. The difficulty this presents is demonstrated by the space devoted to it in (Boyer & Strother Moore, 1981); proofs will typically be by induction over proofs (and require a cut-elimination theorem) or, as here, by induction over computations (and require a normalization theorem). This compares unfavourably with the kind of object-level proofs involved in Howe's and our system.

As with our system, (Boyer & Strother Moore, 1981) uses no separate metatheory, metafunctions and their correctness proofs inhabiting the object level. But as noted above, Boyer and Moore's mechanism has some essentially *meta* parts: their central reflection theorem (the metatheorem) could be neither proved nor expressed in the object logic, nor would meta-definitions be permissible. The correctness theorems as they stand could in fact be used (thanks to the meta-axioms) much as in our mechanism, though without access to meta-definitions and tactics to perform lifting and the other chores of reflection this would be intolerably tedious.

6.2.3 Summary

This is in some ways the most closely related piece of work to ours extensibility via a reflection mechanism and automated correctness proofs. We differ, of course, in that we synthesize tactics whereas (Boyer & Strother Moore, 1981) deals in verification, and in the kind of metafunctions we are trying to produce.

In effect, this work allows one to add to NQTHM any provably correct term transformation tactic, things which would otherwise be impossible to express. Although term rewriting is a very localized procedure, NQTHM's conflation of terms and formulas means that sophisticated behaviour can result, e.g. cancellation, a propositional decision procedure. And while the control available over metafunctions is simple and the overall system heuristic is unchanged, their presence can have a profound effect on the theorem prover's ability.

Any user of NQTHM should be capable of adding metafunctions. Hiding the workings of the reflection mechanism and underlying representation means that a knowledge of how quotations work at the object-level is all that is required. The user of our mechanism will need to be more aware of the underlying machinery, e.g. formulating environments, ensuring monotonicity, using the correct definition convention.

For all the many advantages of the "black box" approach of Boyer and Moore, mainly efficiency related, there are accompanying disadvantages. Among the most important of these is the non-transparency of the reflection process. The user has no control, apart from the function symbol restriction, over how or when the `META` lemma is used. Metafunctions, too, are limited to the form described above; since the user has no direct access to the mechanism he is unable to develop new forms such as the conditional rewrites and others mentioned by Howe (see section 6.3), or the analytic preconditions of Knoblock (see section 6.4). The user also has no control over the implicit reference to the current theory. Finally, there is the problem of portability as discussed in the preceding section.

We remarked above that we found it necessary to go outside the usual forms of definition in OYSTER to simulate some of *CLAM*'s non-monotonic predicates.

This is different from Boyer and Moore's unconventional notion of meta-definition: Howe (see section 6.3) shows that this notion is not essential and can be eliminated by principled use of an explicit environment. Unfortunately we can see no similar means of eliminating our problem.

The comment in (Boyer & Strother Moore, 1981) that use of the CANCEL metafunction slowed the system down by only 0.5% is noteworthy. Although the heuristic control restricts application to only equalities and the metafunction will quickly exit unless it finds PLUS-trees, this is an impressive figure. This statistic and the fact that Boyer and Moore found the correctness proof quite tractable provides strong evidence that efficiently implemented reflection mechanisms are one of the most promising ways of extending a theorem proving system.

6.3 Howe

This section is based on (Howe, 1988a; Constable & Howe, 1990; Howe, 1988b). Chapter 5 of (Howe, 1988a) details Howe's implementation of a partial reflection mechanism which elegantly brings together the ideas of Boyer and Moore (Boyer & Strother Moore, 1981) and Paulson (Paulson, 1983a) in the constructive type theory NuPRL, and it is a simplified, type-less version of this that underlies our work in chapter 3. The mechanism, like that in the preceding section, is a semantic one. However, unlike the preceding section, here the mechanism is built entirely inside the object logic "meta" in the usual sense is a misnomer so that there are no new reflection inference rules or prescriptions on how metafunctions must be used. This logical transparency was one of its attractions for our work. It also ensures that the result is a definitional extension of the original and therefore relatively consistent. Finally, not needing to tamper with the system in any way makes this mechanism much more portable than the others described in this chapter.

We have discussed how our simplified reflection mechanism might be extended to make use of the more sophisticated aspects of Howe's, and how this will in turn

affect reasoning about it, in section ??. To make sense of this we now describe in some detail Howe's work and how it differs from ours. In particular we discuss Howe's use of a *typed* reflection mechanism, the forms of reasoning employed in correctness proofs (especially the major role of evaluation) and their degree of automation, how monotonicity is accommodated, and the distribution of work between the proving and running of a verified tactic.

6.3.1 Aims

Howe's aims in constructing his reflection mechanism are theoretical rather than pragmatic, although aspects of the latter are analysed. These aims are two-fold: an encoding in NuPRL of Bishop's constructive analysis (Bishop, 1967); and providing for the natural expression of new kinds of theorem, e.g. metatheorems which talk about the syntactic properties of terms. A reflection mechanism is also attractive from the tactic writer's point of view for the reasons we give in chapter 3.

The main application of the mechanism is the construction of verified rewriting tactics. Again, the transparency of the mechanism means that one isn't limited to this form of use: Howe also describes an expression normalizer and a more general form of tactic (see below). Like Boyer and Moore, the main question for Howe is "whether it is feasible to formally verify in NuPRL significant procedures for automating reasoning."

Howe's aims are quite different from our own. In our case the reflection mechanism, instead of being an object of study in its own right, is an elegant means of expressing the formal correctness of tactics. The word "verify" in the previous paragraph is with respect to object-level correctness. This is our starting point: the tactics we synthesize must additionally satisfy meta-level criteria arising from *CLAM* method specifications. Howe has nothing like, for example, the analytic preconditions of Knoblock (see section 6.4). Of course, we are also interested in automating large parts of the synthesis proofs and providing a generic high-level story a proof plan.

6.3.2 Technical details

NuPRL is a very expressive constructive type theory. Founded on the programs-as-proofs paradigm (see sections 2.3.5 and A.1), it has an intrinsic higher-order, λ-calculus-based programming language and notion of computation, and many commonly used types are primitive, e.g. atoms, integers, lists, and tuples. There are a couple of conventions in NuPRL of which extensive use is made. The first is the method of definition. As in OYSTER a definition (called `Name`, say) of a term of fixed type T can be defined as the extract of the theorem $>> T$ (called `Name_`). In addition, in NuPRL one may in the same way define a polymorphic object using a theorem of the form `>> object`. An extra, well-typing lemma (called `Name__`) is required in this case. (The naming convention is for the benefit of the well-formedness autotactics.) There is currently no equivalent of the `object` type in OYSTER.

The other convention relates to the computational content of a NuPRL proof. A witness for the assertion $\exists x : T.P$ will comprise the pair $< t, p >$ where t is provably a member of T and p is a witness for the assertion $P[t/x]$. Often one isn't interested in the p part, and the NuPRL *subset* type allows it to be hidden: a witness for the assertion $\{x : T|P\}$ consists only of t, though its proof would similarly require a proof of $P[t/x]$. Precise details can be found in (Constable *et al*, 1986). The subset type is used by Howe to "squash" propositions whose computational contents are uninteresting. It is defined as

$$\downarrow(P) == \{(0 \texttt{ in int})|P\}$$

i.e. it has the sole witness 0 iff P is true.

The most noticeable difference between Howe's mechanism and ours is the presence of metatypes in the former. In line with Bishop's dictum on the properties of set-hood, and because of shortcomings in NuPRL's quotient type, the type `Set`(i) is defined, indexed by universe level. A member A of `Set`(i) is a pair consisting of a carrier type ($|A|$) in `U`i and an equivalence relation over it (written $x = y\{A\}$). As Bishop's analysis does not require it, and because its absence significantly reduces the complexity of defining metatypes, the computational content of the equivalence

relation is squashed. The metatype "Prop" is used to lift top-level assertions and maps to

$$\texttt{Prop} ==< U6, \lambda P, Q.P \longleftrightarrow Q >$$

which is in Set(7) (also called SET). All other atomic metatypes map to types in Set(6) (also called Set). Prop is treated specially throughout the mechanism so that one can always lift equalities in all of the other metatypes. Prop is important since it allows the mechanism to manipulate actual NuPRL *assertions*; it is distinct from Boyer and Moore's *term*-based rewriting, which relies on the fact that propositions masquerade as terms in NQTHM, and our even more restricted mechanism where only terms of type pnat are treated. See also our discussion of this in section ??.

The indexing of universes above is by necessity a simulation and partial since universes are not internally indexed in NuPRL. In this instance indexing is simulated upto to level 12. Thus, the tower of types Set(i) would have had to stop at some point, and this and the predicative nature of NuPRL mean that the reflection mechanism cannot in general be applied to itself. In practice this may not be a very important limitation on "bootstrapping". An identical limitation, of course, exists in OYSTER and the suggestions of section ??. Howe suggests a possibility for avoiding this problem but at the expense of changing NuPRL and a considerable increase in the complexity of the reflection mechanism.

Howe calls his work a "partial reflection mechanism" and there are limitations apart from those on self-application just described. The mechanism's ignoring of computational content is described below, as is its restricted form of type matching. More important is the impossibility of expressing general quantification, or indeed any binding constructs, using Howe's metaterms. Essentially the metalanguage is first-order and quantifier-free.

Howe's raw metaterms, the type

$$\texttt{Term0} == \texttt{rec}(T.(\texttt{Atom}\#T\ \texttt{list})|(T\#T\#\texttt{Atom})|(\texttt{Atom}\#T\#T)|\texttt{Int}).$$

are trees with four kinds of node. Function-application nodes (written "$f(l)$") consist of a function symbol (f) from Atom and a list (l) of Term0 arguments.

Equality nodes (written "x=y in A") consist of two `Term0`'s (x and y) and an `Atom` naming an atomic metatype (A). The third kind of node, i-nodes, are used to represent Bishop subsets in the form of injections; although considerable effort is expended on i-nodes throughout the development of the mechanism no application using them is given, and we shall not mention them further. The final node kind is used to lift integer numerals (written "n"). It can be seen that `Term0` affords linear space and time complexity w.r.t. translation. It is a restricted form of the function-application node that constitutes our metaterms in chapter 3; instead of a variable length list we have a fixed number of arguments.

The dual notions of metatype and metaterm occur at all stages. There are the corresponding type environments and function environments. A type environment maps the names of atomic metatypes to their corresponding object-level sets. More formally, an element of the type `TEnv` is an association list (a-list) of `Atom#TEnvVal` pairs. The `Atom` is the name of the atomic metatype, and the `TEnvVal` the member of `Set(6)` to which it is mapped, plus an indication of whether its equality is *trivial* (i.e. has no computational content). Metatypes may have the form

$$\beta_1 \# \cdots \# \beta_n \rightarrow \beta_0$$

where $n \geq 0$ and $\beta_0, \ldots, \beta_n$ are atomic metatypes. When $n = 0$ this is treated as the type β_0. Thus, although the underlying sets may not be, at the meta-level types are restricted to this first-order form. Well-formedness similarly restricts function-application nodes (see below). In keeping with "`Prop`"'s special treatment it is always mapped to `<Prop,fail>` `fail` indicates that `Prop`'s equivalence relation is non-trivial.

Function environments are defined w.r.t. a given type environment. They too take the form of a-lists, but of `Atom#FEnvVal(`γ`)` pairs, where γ is a type environment and `FEnvVal(`γ`)` consists of: a metatype as described above and represented as the pair $< [\beta_1, \ldots, \beta_n], \beta_0 >$, an object-level function, and an indication of the nature of this function (principally its computational content which, along with information from the type environment on the triviality of a metatype's equality, is used when "squashing" a lifted sequent (see below)). An environment is then simply a

pair consisting of a type environment and an appropriate function environment:

$$\texttt{Env} == \gamma : TEnv\#FEnv(\gamma)$$

Howe defines an initial environment $\alpha_0 ==< \gamma_0, \delta_0 >$ on which all others are built. This has the type environment γ_0:

```
"Int"   ↦  <↑(Int),s(Int_eq_triv)>
"False" ↦  <↑(False),s(False_eq_triv)>
"True"  ↦  <↑(True),s(True_eq_triv)>
```

and the function environment δ_0:

```
"True"  ↦  <<nil,"Prop">,True,no_kind>
"False" ↦  <<nil,"Prop">,False,no_kind>
```

↑ (T) denotes the canonical injection of a NuPRL type T with its usual computational content-free equality into `Set`(i). The types `False` and `True` are synonyms for the types `void` and `(0 in int)`; they are also (nullary) functions of type `Prop` (see above).

The metaterms of our chapter 3 *always* map to values in the OYSTER type `pnat`. Like Boyer and Moore, symbols which aren't specified by the environment are mapped to a default value. The case with Howe's mechanism is much more complex. To start with, not all members of `Term0` are well-formed. To take function-application nodes as an example, both the number and metatypes of the argument list `"l"` must "match" the metatype of `"f"`, the metatypes in each case being provided by the function environment, if `"f(l)"` is to make sense; and the arguments must themselves be well-formed. To this Howe adds the requirement that the matching of metatypes must be more flexible than mere identity more like the behaviour at the object level. The result is that the definition of well-formedness depends upon that of evaluation of metaterms. But evaluation is only defined for well-formed terms. In NuPRL this mutual recursion requires a complicated simultaneous induction proof on terms of type `object` before a properly typed definition is possible.

At this point the following functions are defined with the following properties (compare this with our single function `eval` in chapter 3). For raw term $t \in Term0$ and w.r.t. to the environment α:

`wf@(`α`,t)` says that `t` is well-formed

`Term(`α`) == {t:Term0 | wf@(`α`,t)}` is the set of well-formed terms

`mtype(`α`,t)` gives the ostensible metatype of `t` $\in$ `Term0` based on its outermost constructor.

`type(`α`,t)` gives the object-level member of `SET` corresponding to the metatype of `t`, i.e. it is the evaluator for types

`val(`α`,t)` gives the object-level member of `type(`α`,t)` corresponding to `t`, i.e. it is the evaluator for values (the equivalent of our `eval`)

Making sure that all these are well-formed is the central theorem of the reflection mechanism which states (note that well-formed terms must have atomic types):

$$\forall \alpha : \mathtt{Env}, t : \mathtt{Term0}.\mathtt{wf@}(\alpha, t) \ \mathtt{in}\ \mathtt{U} \wedge \downarrow(\mathtt{wf@}(\alpha, t)) \rightarrow$$
$$(\mathtt{mtype}(\alpha, t) \ \mathtt{in}\ \mathtt{AtomicMType}(\alpha) \wedge \mathtt{val}(\alpha, t) \ \mathtt{in}\ |\mathtt{type}(\alpha, t)|) \qquad (6.1)$$

As explained in chapter 3, a usable mechanism must be monotonic. Howe takes a different approach to ours: although like ours the necessary objects are monotonic, often the requirement will be an explicit part of the definition. For example, using the notation $\mathtt{cst}(\alpha_1, \alpha_2)$ to denote consistency of environments, and $\alpha_1 @ \alpha_2$ their join, the definition of the type of rewrite tactics is:

$$\mathtt{Rewrite}(\alpha) == \{f : \mathtt{Term0} \rightarrow \mathtt{?Term0} | \forall \alpha_2 : \mathtt{Env\ where\ cst}(\alpha, \alpha_2).\mathtt{val\_inv}(\alpha @ \alpha_2, f)\}$$

I.e. members must be valid not only in α but in all extensions of it. The main monotonicity result is then an immediate consequence:

$$\forall \alpha_1, \alpha_2 : \mathtt{Env}.\alpha_1 \subset \alpha_2 \rightarrow \forall f : \mathtt{Rewrite}(\alpha_1).f \ \mathtt{in}\ \mathtt{Rewrite}(\alpha_2)$$

This form of definition also illustrates another important aspect of the development of the mechanism. Howe has relied extensively on the use of exhaustive evaluation

as a means of proof. Though simple when it works, it can lead to some difficulties when defining concepts. Howe mentions in his conclusion that one definition lead to "serious complications" when it was noticed too late that it did not allow this style of proof. It can also lead to less general and declarative definitions. Throughout, for example, Howe uses the "trick" of proving lemmas of the form $\forall\alpha : \mathtt{Env}.P(\alpha_0 @ \alpha)$ instead of the more usual monotonicity lemma $\forall\alpha : \mathtt{Env}.\alpha_0 \subset \alpha \rightarrow P(\alpha)$. ($\subset$ is the sub-environment relation.) The advantage of the former is that when, as will often be the case, α_0 is concrete but α is abstract (e.g. a variable), evaluation is still possible since list recursion proceeds head-first. We have not taken this approach.

For all the above, it should be noted that the monotonicity of Howe's mechanism is completely explicit. Again, this confers transparency and portability on the mechanism, as well as providing a powerful abstraction mechanism. Compare this with the unprincipled part of our use of load-time simulation, and Boyer and Moore's somewhat circuitous and highly system-specific metatheoretic demonstration of monotonicity.

The lifting of a sequent is a multi-stage process. As with our version, the user is required to specify the environment to be used for lifting, and any variables present in the sequent at the time must additionally be lifted. Howe's lifting tactic (`LiftUsing`) takes a list of environments from the user. Clearly, the reflection mechanism's expected mode of use is the development of small, disjoint (there is a special check for this for efficiency's sake) environments which are combined at lift-time. Thus lifting becomes quite an expensive operation, requiring a check for pairwise consistency of all the environments and the cutting in of a hidden hypothesis asserting that each is a sub-environment of the combined one. This is quite different from our approach where environments are treated like theories and combined statically at load-time via "bridging" lemmas. Howe's method has the advantage that it is easy to add small changes as a library of proofs is developed, but the disadvantage that there is a large overhead involved in joining and proving consistent a number of environments each time a reflection proof is performed. Ours is the reverse: at the cost of forcing the user to bundle library objects

into environments as they are completed, the cost at the time of reflection is greatly reduced. This is appropriate in our case since the same synthesized tactics, corresponding to $CLAM$'s methods, will be re-used many times.

The lifting-transforming-lowering sequence of a reflection proof is like that described in chapter 3, with the addition of an extra layer to handle types and formalized in the central typing lemma (6.1). However, unlike both Boyer and Moore and us, the lifted sequent is not merely a transient intermediary. Much information relating to the environment and well-formedness of the metaterms is hidden in an elided hypothesis. Along with a conscious effort to make lifted terms resemble as closely as possible the objects they represent, this means that in use the lifted sequent appears like a normal sequent with metafunctions taking the place of inference rules, and that several metafunctions in succession may be applied before the result is lowered.

An attempt is made by the lifting tactic to lift the conclusion and all (non-declaration) hypotheses of the sequent. Metaterms are computed by lifting an object-level function application term of the form `term_of(f_)(a1)...(an)` to `"f([b1,...,bn])"`, where `bi` is the lifted form of `ai`. Like ours, this restriction to functions defined in the "term_of" style does not limit the mechanism since any definition can easily be given this form. The sets appearing in equality metaterms are lifted in the same way.

Because of his decision to ignore the computational content of propositions it is in general not possible to rewrite arbitrary assertions with Howe's reflection mechanism. Therefore an attempt is made to squash the conclusion after lifting, using information from the environment about the kinds of the equalities and functions present. (Although the mechanism is not limited to it, Howe's tactics are oriented towards rewriting only the conclusion of a sequent.) If the attempt at squashing fails, lifting fails. If successful, a sequent of the form

$$H_1, \ldots, H_n >> G$$

will after lifting become

$$\alpha : \mathtt{Env}, (\ldots), \mathtt{val}(\alpha, \text{“}H_1\text{”}), \ldots, \mathtt{val}(\alpha, \text{“}H_n\text{”}) >> \downarrow (\mathtt{val}(\alpha, \text{“}G\text{”}))$$

where (...) is the elided hypothesis mentioned previously. For efficiency reasons the combined environment, $\overline{\alpha}$ say, is abstracted to a variable α, and the hypothesis $\alpha = \overline{\alpha}$ `in Env` elided. $\overline{\alpha}$ will typically be a very large term. This abstraction means that it need only be substituted when required to carry out evaluation. Lowering a lifted sequent is achieved by simple evaluation, as in our case.

Howe's metafunctions for the most part are members of the type `Rewrite`(α) (see above). These are applied to the lifted sequent by a specific tactic, `RewriteConcl`, taking a member $f \in$ `Rewrite`(α) as its argument. If the result of evaluating $f[\overline{\alpha}/\alpha]$ applied to "G" is a failure injection then the rewriting is aborted; otherwise, for a success injection of "G'", the conclusion becomes $\downarrow$ (`val`$(\alpha$, "G'")) and the elided hypothesis is updated accordingly. The form of evaluation used here appears to be a very efficient, low-level NuPRL routine (`eval`) not immediately available as a primitive inference rule, specially introduced by Howe for this purpose; it can be simulated though much less efficiently using the existing direct computation rules and appears to have become an established feature of NuPRL. Our approach to metafunction application is identical to that just described, though we have no equivalent of `eval` in OYSTER.

Showing a function f to be a member of `Rewrite`(α) is very similar to the requirements of Boyer and Moore and to ours. Proving that $f \in$ `Term` $\rightarrow$ `?Term0` is equivalent to showing their `(FORMP (fn X))`, and the `val_inv`$(\alpha @ \alpha_2.\ f)$ part to their `MEANING`-invariance requirement. Note that `val_inv` is defined to have no computational content and so that metatypes must match exactly, i.e. if $f(t)$ evaluates to `inl`(t'), then `val_inv`(α, f) is:

$$\downarrow (\texttt{mtype}(\alpha, t) = \texttt{mtype}(\alpha, t')\ \texttt{in Atom} \wedge \texttt{val}(\alpha, t) = \texttt{val}(\alpha, t')\ \texttt{in type}(\alpha, t))$$

We have already mentioned the most important method of using the reflection mechanism: rewriting of the conclusion using members of `Rewrite`(α). We too have adopted Howe's convention of making the notion of failure explicit, i.e. when an $f \in$ `Term0` $\rightarrow$ `?Term0` is applied to a metaterm $t \in$ `Term0`, the output is a left injection of the result if successful, and a right injection if the application failed. Howe develops this idea much more thoroughly than we have, defining failure-

aware analogues of most of the common constructs used elsewhere in the library. The most important use of failure is in allowing the easy construction of complex tactics from simpler ones using Paulson-style combinators. Our work utilised only a subset of Howe's: `Repeat`, `ORELSE`, `THEN`, `Try`, `Progress`, `Sub`, `TopDown`, `BotUp`, and `Topmost`.

Of particular interest is `Repeat`. We encountered the same need for general recursion, and therefore partial types, if one is faithfully to implement this tactical. Since these were not available in NuPRL at the time of (Howe, 1988a) a work-around was implemented. This entails approximating the fix-point of a function (the standard way for defining recursive functions in the λ-calculus) by a fixed, though very large, number of iterations. This number is so large that due to physical constraints there will be no discernible difference in the behaviours of the true and "fake" fix-point. However, their behaviours are not provably the same. This distinction is not important to Howe since he is not interested in the specifications of his tactics (other than that they belong to Rewrite(α)). We have discussed our solution to this problem in chapters 3 and 5. In short, although we require our tactics to provably satisfy their specifications, atomic *CLAM* tactics are sufficiently well instantiated by the time they come to be executed that any kind of search (which would require such general recursion) can be excluded — it can be removed to the Prolog level.

Making use of the powerful NuPRL membership tactic and a small number of lemmas showing that the combinators preserve membership of `Rewrite`(α), e.g.

$$\forall\alpha : \texttt{Env}.\forall f, g : \texttt{Rewrite}(\alpha).f \texttt{ THEN } g \texttt{ in Rewrite}(\alpha)$$

verifying that complex tactics are genuine metafunctions usually amounts to no more than simple, unaided type-checking.

Perhaps more interesting from the point of view of automating the synthesis of tactics is how the atomic tactics come about in the first place. Rather than synthesize them from starting specifications as we have, Howe "lifts" already proved, object-level theorems having the form of a universally quantified equation. To go from an equation to a rewrite procedure requires an object-level matching algorithm.

This is developed by formalizing the notion of various classes of substitution and *first-order matching*, culminating in the constructive existence theorem (producing the substitution s):

$$\forall vars : \texttt{Atom list}.\forall t_1, t_2 : \texttt{Term0}.?(\exists s : \texttt{Atom\#Term0 list where}$$
$$(s\{vars\}\ \texttt{complete on}\ t_1) \wedge s(t_1) = t_2\ \texttt{in Term0})$$

from which the matching algorithm is extracted. As with most of our mechanism we have followed Howe in this, though we have used it slightly differently: we use matching to simulate CL^AM's use of Prolog's unification when applying various rules, some of which are equivalent to rewriting with an equation. We have also formalized the notion of pattern variable differently, having a dedicated node for this purpose in our definition of metaterm; Howe uses an explicit list of variable symbol names (*vars* in the theorem above).

Howe gives several examples of verified rewrites. These are members of $\texttt{Rewrite}(\alpha\_\texttt{Q})$, where $\alpha\_\texttt{Q}$ is an environment defining the rational numbers and the addition and multiplication operations on them. In one example the object-level equation

$$\forall x, y : |\texttt{Q}|. -(x + y) = -x + -y\ \texttt{in Q}$$

is lifted and combined with the tacticals `Repeat` and `TopDown` to produce a rewrite which exhaustively moves "$-$" signs outwards over "$+$" signs; its membership of $\texttt{Rewrite}(\alpha\_\texttt{Q})$ is shown completely automatically. Another example is an instance of the expression normalizer described below.

It is interesting to contrast the different methods used to construct proofs in our and Howe's mechanisms; this is apart from the methods used to synthesize tactics. An earlier chapter of (Howe, 1988a) covered the development of a principled tactic library for NuPRL. This depends heavily on the definition convention described above, and also on the user's carefully updating global lists, or, like Boyer and Moore, making sure appropriate lemmas are present. (It is notable that the majority of these lemmas are ordinary or higher-order *wave rules*.) These lists are used by the autotactics when their default behaviour is likely to be insufficient to complete a proof. Examples are autotactics for proving well-formedness, membership, type inclusion and decidability. A common proof method of Howe's is

exhaustive evaluation in the hope that a term will simplify to, say, `True` or `False`. With the increasing complexity of the theorems in the reflection library this approach often fails. A strategy for overcoming this is typified by the `type_atom` "suite".

First the declarative, "real" definition is given, called `type_atom@` and, as usual, consisting of the theorem `type_atom@_` and the definition `type_atom@`. This definition, though clear in meaning, is either not executable or hopelessly inefficient. It is followed by a theorem (`type_atom@_decidable`) showing that the the defined property is decidable. Some ML code then adds this fact to a global list as mentioned previously. Next, what might be called the procedural version of the definition, one that can be efficiently evaluated, is defined using theorem `type_atom_` and definition `type_atom`; it is usually based directly on the preceding decidability proof, and a totally ground instance should evaluate to either `True` or `False`. Finally, a characterization theorem (`type_atom_char`) is given which shows that the two definitions are equivalent, in this case:

$$\forall \gamma : TEnv, a : Atom.type\_atom(\gamma, a) \leftrightarrow type\_atom@(\gamma, a)$$

Such theorems can be used to replace "declarative" statements with equivalents which are more amenable to proof by evaluation.

It may not always be feasible to prove a decidability lemma as in the above case. Where a full decidability theorem is not required Howe often proves an underspecification of the problem: instead of showing $P \vee \neg P$, a theorem showing $?(P)$ is proved. While such a theorem can be trivially proved, a convention is used whereby a left (success) injection occurs in exactly the cases where P is true. Alternatively, an efficient characterization (such as `type_atom_`) may be still be possible, *most* of which can be evaluated away, e.g. those properties for which a decidability lemma is present or which the decidability tactic can prove. Howe uses an interesting method to ensure that evaluation of such terms proceeds as thoroughly as possible. A type of *partial booleans* is defined

$$PBool == Bool \vee U$$

which allows for injections of known boolean values or arbitrary propositions $\in U$. Analogues of all the logical connectives, including various forms of bounded quantification, are defined so that whenever the arguments contain sufficient information to determine a boolean value, that is the result; otherwise a general proposition results. When procedural characterizations of definitions are written using these `PBool` forms of the connectives it is often possible to remove all the decidable components of the expression simply by evaluation. Use of the `PBool` type is made relatively transparent by the definitions mentioned, characterization lemmas relating them to the `Bool` forms, and special-purpose tactics.

Compare the above approach with our use of *CLAM* when developing part of the mechanism. The only autotactic used is for well-formedness subgoals (which *CLAM* does not examine in any case). Instead of a decidability tactic, for example, *CLAM* was used to plan decidability proofs using only definitions and a small, constant set of chaining lemmas (see section 4.6). Of course, in other contexts exhaustive evaluation was often the most effective means of proof.

As mentioned previously, one of the attractions of Howe's mechanism over Boyer and Moore's was its transparency, it's core implementation being completely object-level. Essentially all it does is provide a mapping from metaterms to the objects they represent. Howe demonstrates the flexibility this engenders with two further quite different forms of meta-procedure.

To define rewriting meta-procedures Howe used the type $Rewrite(\alpha)$. He defines a more general class of meta-procedures which apply to the whole of a lifted sequent, the hypotheses being represented as a list of metaterms:

$$Complete(\alpha) == \{t : PropTerm(\alpha) | t(\alpha)\} list \rightarrow concl : PropTerm(\alpha) \rightarrow ? \downarrow (concl(\alpha))$$

where $PropTerm(\alpha)$ defines a class of metaterms whose metatype is `"Prop"`, and $t(\alpha)$ abbreviates $val(\alpha, t)$. A different tactic (`ApplyCompleteTac`) is used to apply members of $Complete(\alpha)$. Howe gives the example of an equality procedure which decides whether the equality in the conclusion follows from equalities amongst the hypotheses and the laws of symmetry and transitivity.

This is very interesting from our point of view since we have already noted the limitations imposed by our choice of synthesizing only rewrite tactics, in particular the impossibility of a fertilization tactic in such a setting. *Complete*(α) suggests a way of proceeding to sequent-level tactics with little extra work or added complication to synthesis proofs; we comment further on this in chapter 7.

The second example of Howe's is an expression normalizer. This takes terms in a commutative monoid (whose definition Howe formalizes) and produces a normal form by sorting the components and eliminating the identity operator where possible. The order used to perform the sort is taken from the order in which declarations occur in the hypothesis list and is available to the metafunction via its environment argument. Howe shows that the normalizer is a member of `Rewrite`(α) when α includes a suitable commutative monoid.

6.3.3 Summary

It was explained above that in many places in the development of Howe's reflection mechanism, to keep what is already complex manageable, the subset type is used in place of the product type. In particular the decision to ignore computational content in the definition of *Set*(i) and in the definition of value-invariance as part of *Rewrite*(α) ensures that only terms without computational content may be rewritten. This is unfortunate since the important uses of constructive type theory (and goals of proof planning) include formal program verification and synthesis, where computational content *is* significant. Although this consideration is immaterial in our simplified mechanism, we feel it to be a prerequisite for a full-scale system for tactic synthesis. This is discussed in more detail in section ??. Note that the notion of preserving computational content is not applicable to NQTHM and Boyer and Moore's work.

As we have noted already in several places, while Howe and Boyer and Moore both have the goal of sound extensibility, their approaches are quite different in emphasis — one could say theoretical elegance vs pragmatism. Howe notes that there are possible problems with his mechanism in practice, for example the time

taken to lift and drop sequents (typically > 90% of the expense of a reflection proof), problems with the continual reproving of well-formedness subgoals (an inherent problem of the NuPRL type theory), and the expense of maintaining the hidden hypothesis in a lifted sequent. He also notes that running NuPRL programs is quite inefficient but that this may not be a fundamental problem, related more to the naive and unoptimized evaluation algorithm and NuPRL's being written in an old implementation of ML. Boyer and Moore have shown that very efficient reflection mechanisms are possible (see section 6.2). Howe also does not indicate a means of using meta-procedures as an everyday part of NuPRL, for instance how they might be used as standard lemmas are by the various autotactics. In contrast, Boyer and Moore concentrate almost wholly on making their mechanism very efficient and hiding the details from the user so that a metatheorem, once proved, is seamlessly incorporated into NQTHM's "black box" and used much like any other lemma. It would seem that this trade-off between transparency of mechanism and ease of use is hard to avoid.

Howe considers that there remains too the problem of inflexibility of type-matching at the meta-level in his mechanism, which was mentioned briefly above. To achieve flexibility, however, one must choose between two problematic courses: perform most of the type-matching at run-time, thereby undoing one of the main attractions of using reflection in the first place ("proof compilation"); or reflect a large part of the object-level typing rules, thereby losing the attractive simplicity of semantic reflection. This is likely to remain a problem in non-strongly-typed theories such as NuPRL. In the proofs for which $CLAM$ has been used upto the present, type flexibility is not important.

A significant difference between our mechanism and Howe's is our need to refer to meta-level data. For example, because $CLAM$ uses, say, the predicate `func_defeqn/3` to look up defining equations, our tactic specifications, and the resulting synthesized tactics, must interact with an analogue at the object-level (`def_eqn`). This leads to the problems with ensuring monotonicity described in sections 3.4.2 and 3.5. Howe's metafunctions, on the other hand, apart from the requirement that they be value-invariant, have to satisfy no other specification at all. Unlike Howe

we also use meta-level annotations in our lifted terms, such as those indicating wave fronts and holes, and synthesize tactics which act upon and manipulate them.

Besides this difference in the nature of tactic synthesized and Howe lifts his atomic tactics from already proved equations rather than synthesizing them from specifications as we do we also differ from Howe in the kind of automation employed to help with proof. Howe has developed several powerful tactics, and clever though specific techniques such as use of the "$\alpha_1 @ \alpha_2$" form and partial booleans, which enable large and tedious parts of the proof to be dispensed with. It is noticeable though that instances of induction must be done by hand. We instead have tried to tell a high-level story using *CLAM* and proof plans, particularly for induction proofs, using methods which apply across a wide variety of contexts and logics.

We have shown the need for at least a restricted form of higher-order rewriting in section 4.8, and quantifier manipulation is quite extensive in present-day *CLAM* methods. Thus a mechanism capable of fully reflecting current *CLAM* proofs will need a metaterm syntax considerably richer than Howe's. If we are eventually to synthesize tactics such as *CLAM* presently uses then a reflection mechanism at least comparable to Howe's will be necessary. The problems outlined in this section therefore remain to be solved to achieve this goal.

6.4 Knoblock

This section is based on (Knoblock, 1987; Knoblock & Constable, 1986) in which three reflective systems are studied.

Like Howe, Knoblock attempts to formalize part of, in this case nearly all, the NuPRL logic. In contrast to Howe, who does this *inside* NuPRL by partial self-reflection, Knoblock augments NuPRL with rules describing the proof theory of NuPRL to obtain a separate metatheory he calls Metaprl. This is syntactic reflection. Calling NuPRL PRL^0 and Metaprl PRL^1, Knoblock generalizes the idea

to obtain a hierarchy PRL^i, for $i \in \omega$, where each theory is the metatheory of its predecessor. In (Knoblock & Constable, 1986) a third system whereby PRL^0 is partially reflected into itself via a reflection principle is also discussed.

All three systems are of interest since each is capable of supporting tactic synthesis. Indeed, one main motivation of this work is to replace the informal meta-language ML of NuPRL with a formal one, Metaprl, very much like NuPRL itself. If this goal were achieved one would have a unified language for object- and meta-theory, and the attendant reduction in duplicated effort. It is Knoblock's intention that all things previously external to NuPRL proofs, such as tactics, the library mechanism, definition mechanism, and certain anomalous inference rules, be carried out instead in Metaprl.

The other main goal of this work is to raise the level of reasoning at which proofs are conducted. The main idea here is to prove that satisfaction of an "analytic condition" possibly in quite abstract and high-level terms akin to normal mathematical discourse is sufficient for the successful application of a formal tactic. Since these conditions, in the form of *applicability predicates*, may be much cheaper to compute than actually carrying out a primitive proof (see p. 3 for an example), forms of reasoning which are impractical using the traditional tactic-based approach (which must in the end perform proof at the primitive inference rule level) may come into scope. This notion is reminiscent of a formal version of proof planning and would serve as an elegant foundation for our work. However, while (Knoblock, 1987) gives a precise description of Metaprl it appears that no implementation exists.

Since Knoblock's goal is to formalize anything a tactic is capable of, a full, reflected proof theory of NuPRL is required. Since it is not possible to do this in NuPRL itself and remain consistent, a separate metatheory, Metaprl, is used. Metaprl contains all the terms and rules of NuPRL plus new rules specific to the metatheory. More precisely, Knoblock adds the new types `char_string`, `ident`, $\texttt{term}^0$, $\texttt{rule}^0$ and $\texttt{proof}^0$, which represent their NuPRL namesakes, and the new operators $\texttt{applies}^0(\cdot,\cdot)$ and $\texttt{subgoals}^0(\cdot,\cdot)$, which represent NuPRL proof refinements. All other necessary notions can be defined in terms of these.

The reflection mechanism is centred upon the idea that irreducible members of the Metaprl type $\texttt{proof}^0$ are in one-to-one correspondence with complete primitive NuPRL proofs. Thus one has a bi-direction reflection principle of the form

$$\frac{>>^1\texttt{proof\_of}^0(\text{“}\sigma\text{”})}{>>^0\sigma}$$

where $>>^i$ is the PRL^i turnstile and $\texttt{proof\_of}^0(s) = \{p : \texttt{proof}^0 | \texttt{goal}^0(p) = s \texttt{ in sequent}^0\}$. It is not clear whether the principle has the status of a formal rule. In (Knoblock, 1987) Knoblock says that "it is the system that implements the correspondence" and that switching between the two levels is carried out from within the proof editor, that "NuPRL proofs as conventionally written are just an alternative syntax, a different notation for constructions in the type $\texttt{proof}^0$ in Metaprl." This last suggestion resembles Boyer and Moore's idea whereby terms and their quotations have identical representations as long as one remembers which level is being referred to. We discuss this further in section 7.2.2.

The validity of the reflection principle above is shown in (Knoblock, 1987) by defining successively the obvious bijections between NuPRL terms and canonical members of $\texttt{term}^0$, rules and $\texttt{rule}^0$, and proofs and $\texttt{proof}^0$. This leads to proofs that Metaprl is *adequate* that if π is a NuPRL proof of σ then there is a corresponding Metaprl term $\overline{\pi} \in \texttt{proof\_of}^0(\text{“}\sigma\text{”})$ and *faithful* for every Metaprl term $p \in \texttt{proof\_of}^0(\text{“}\sigma\text{”})$ there is a NuPRL proof π such that $\overline{\pi} = p \in \texttt{proof\_of}^0(\text{“}\sigma\text{”})$ for NuPRL, thus justifying the reflection principle in each direction. Metaprl is also shown to be consistent relative to NuPRL by defining a function embedding the former in the latter and showing that, under this mapping, Metaprl is a conservative extension of NuPRL.

Having formalized the notion of primitive NuPRL proof, Knoblock turns to formalizing tactics: functions which construct proofs. He divides these into three classes: complete, partial and search tactics.

Complete tactics correspond to derived axioms of NuPRL in the sense that they construct complete proofs, i.e. with no remaining open subgoals. They are classified in terms of their applicability predicates P so that

$$\text{c\_tactic}^0(P) := s : \{s : \texttt{sequent}^0 | P(s)\} \rightarrow \texttt{proof\_of}^0(s)$$

These correspond most closely to our synthetic tactics, the method pre-condition constituting the applicability predicate.

Partial tactics correspond to derived rules of NuPRL in the sense that they construct proofs which may contain open subgoals. This requires a notion of partial proof and, for reasons of efficiency computing open subgoals, a notion of validation similar to LCF's (see section 2.2). Thus we have:

$$\begin{aligned} \texttt{prf\_of}^0(s) &:= g : \texttt{sequent}^0 \texttt{ list} \times \texttt{validation}^0(g, s) \\ \texttt{validation}^0(g, s) &:= \texttt{proofs\_of}^0(g) \rightarrow \texttt{proof\_of}^0(s) \end{aligned}$$

and the type of partial tactics:

$$\texttt{p\_tactic}^0(P) := s : \{s : \texttt{sequent}^0 | P(s)\} \rightarrow \texttt{prf\_of}^0(s)$$

The applicability predicates of complete and partial tactics are used in the same way. Knoblock envisages that these will typically be decidable, as we do, and that the system will have available proofs of this whose extracts can be computed to determine whether a tactic is applicable to a given sequent. If there is no such proof then the onus for proving $>>^1 P(\text{“}\sigma\text{”})$ falls to the user. If a tactic is found to apply then in the case of a complete tactic, unless an explicit proof is required rather than merely the guarantee that one could be constructed, no more work is required. The case for partial tactics is similar except that the list of subgoals (g) will also need to be computed.

Search tactics correspond to the traditional LCF notion of tactic: they attempt to directly construct proofs and signal failure when something goes wrong. The usual sum type representation of failure is used, allowing combination via analogues of the traditional tacticals (see (Howe, 1988a) and chapter 3). These tactics have the type:

$$\texttt{s\_tactic}^0 := s : \texttt{sequent}^0 \rightarrow ?\texttt{prf\_of}(s)$$

The rationale behind search tactics is that for some proof procedures there may be no better way of deciding success than what amounts to building a proof. Thus

there is no advantage to be gained from putting this check into an applicability predicate; there may even be a duplication of effort if one had to anyway construct a proof later.

(Knoblock, 1987; Knoblock & Constable, 1986) are intended as foundational studies and few applications are given. However, Knoblock does show how tactics corresponding to primitive refinement rules can be easily built as well as a simple well-formedness tactic. It is also shown how partial proofs can be formally grafted together and, hence, how complete and partial tactics may be composed. In this way the synthesis of NuPRL tactics becomes theorem proving in Metaprl. By using a suitably modified proof editor one can build partial tactics by "emulating the proof that the [corresponding] rule is derived."

It will be noticed that tactics as defined above make no reference to an environment argument. As was mentioned, Knoblock intends that the library and definition mechanisms and use of lemmas be carried out in Metaprl so that members of $\texttt{proof}^0$ are stand-alone, type theoretic truths. Yet tactics which operate at a high or abstract level could be expected to refer to a library and in this sense to be environment-dependent, but it is never made clear how this is to be formalized in Metaprl. In other words, monotonicity, while likely to be a problem, is never discussed. This problem is addressed in later work on Metaprl by different authors (Allen, 1990).

We have already mentioned the two main advantages of Metaprl: a unified object- and meta-language; and that in many cases one need not construct an explicit proof, merely show instead that the applicability predicate is satisfied. The latter of course requires that one has proved beforehand that a putative tactic is a member of $\texttt{c\_tactic}(P)$ or $\texttt{p\_tactic}(P)$, this burden being similar to that of the other correctness proofs mentioned in this chapter.

The cases mentioned above where there may be no need to build a proof (i.e. compute an irreducible member of $\texttt{proof}^0$) are when the computational content, if any, is not required. To this end Knoblock classifies the premises of formalized inference rules as *computational* or *non-computational* according to whether their extracts appear in the extract of the conclusion, and extends the classification to

branches of proofs. Thus, when extracting a witness from a complete proof only the computational branches need be constructed. This leads Knoblock to suggest that tactic reduction should be lazy resulting in "proof on demand". This idea is particularly attractive for well-formedness tactics since such proofs are non-computational and are an important factor affecting practical use of NuPRL (see Howe's comments in section 6.3).

The reasons for NuPRL's requiring Metaprl apply equally to Metaprl itself, and so on. Thus Knoblock generalizes the idea to a cumulative hierarchy of formal type theories, PRL^i, where one has roughly:

$$\begin{aligned} \mathrm{PRL}^0 &= \text{NuPRL} \\ \mathrm{PRL}^{i+1} &= \mathrm{PRL}^i + \text{metatheory of } \mathrm{PRL}^i \end{aligned}$$

The metatheory is generalized in the obvious way so that one has new types $\texttt{term}^i$, $\texttt{proof}^i$, $\texttt{c\_tactic}^i$ etc. in PRL^j, and the reflection principle

$$\frac{>>^j \texttt{proof\_of}^i(\text{“}\sigma\text{”})}{>>^i \sigma}$$

for $j > i$.

With all PRL^is being so similar the reasoning methods employed in each are correspondingly similar. However, there is the serious problem that each PRL^i requires its own set of tactics. For example, a member of $\texttt{proof}^1$ will not in general be a member of $\texttt{proof}^0$. Knoblock shows how to trivially "lift" tactics by strengthening the applicability predicate or simulating higher level proofs, but neither of these approaches solves the general problem. Knoblock conjectures that this may not be a hindrance in practice since only the first few levels of the hierarchy will ever be used, e.g. one could prove all tactics in PRL^4, say, and use them at all levels below.

Acknowledging the "intellectual burden" of working in a hierarchy, and perhaps the problem of generalizing tactics across levels, (Knoblock & Constable, 1986) suggests a third approach of partial self-reflection of NuPRL. Since a general self-reflection principle is not possible while retaining consistency, Knoblock finds it

convenient to stratify the principle according to proof level[7]. $\texttt{proof}_i$ is defined as a subtype of $\texttt{proof}^0$ which represents level i proofs, and the other new types and functions of Metaprl are stratified in the same way. The main result of (Knoblock & Constable, 1986) is a metatheorem justifying a self-reflection principle. However, only an outline of the proof is provided and (Howe, 1988a) states that it was never completed. In (Knoblock, 1987) the self-reflection approach is deemed "too complicated and cumbersome for practical use," and appears to have been abandoned.

6.4.1 Summary

(Knoblock, 1987; Knoblock & Constable, 1986) are theoretical studies and are not concerned directly with tactic synthesis or the automation of this. The work is relevant to ours insofar as it provides an elegant framework for a much more general and complete form of tactic synthesis than we have attempted; the use of an applicability predicate influenced the eventual form of our tactics. However, we have mentioned above that the lack of any mechanism for dealing with monotonicity is a serious practical drawback affecting especially the extent to which abstraction of tactics is possible.

While Knoblock's reflection of a full proof theory for NuPRL has the obvious attractions of completeness and generality when compared against our use of semantic reflection, the very complexity of NuPRL over 100 inference rules, complicated notion of proof militates against it. This may also be a factor in the non-implementation of Metaprl and would certainly affect the task of automating synthesis. There is also the question, common to our and Howe's work also, of whether NuPRL is a suitable programming language for tactics. Knoblock's view, like ours, is that only "a complete implementation and extensive use" can test the hypothesis, but that "NuPRL logic provides an excellent framework for metamathematical extensibility."

---

[7] A proof whose greatest occurring universe is less than `Ui` is called a *level i proof*.

6.5 Weyhrauch and FOL

This section is based on (Aiello & Weyhrauch, 1980; Weyhrauch, 1980; Weyhrauch, 1982). While this work hasn't directly influenced ours, several ideas originate here which have been taken for granted by later work in the area. (Weyhrauch, 1980), for example, is the earliest example of an *implemented* reflection mechanism, showing both the feasibility and attractiveness of the idea. It is important from the AI standpoint in emphasizing that reasoning and knowledge representation should take place in a metatheory as well as the object theory.

(Weyhrauch, 1980), as an informal "ideas" paper, is more concerned with laying out the main aims of FOL simulating aspects of human reasoning, the importance of LS pairs for knowledge representation, use of a metatheory and reflection principles, and self-reflection in the structure META than with correctness. The latter is addressed to some extent in (Aiello & Weyhrauch, 1980) and (Weyhrauch, 1982). FOL has been re-implemented and the ideas extended in (Giunchiglia & Traverso, 1990).

FOL is a classical first-order predicate calculus proof checker with some interesting additions. Logical theories are represented as *LS pairs* (later called *contexts*). L, the language, is the usual declaration of syntax. S, or SS, the *simulation structure*, is the mechanical analogue of a model of L. The latter consists of a set of algorithms, here in the form of LISP routines, which may by the act of *semantic attachment* at the discretion of the user give an "effective" rendering of the domain, function and predicate symbols of L. LS "pairs" also have a third component, F, consisting of the facts or axioms of the theory. Semantic attachment is compounded with rewriting by a set of equations/equivalences to provide *evaluation* in FOL. The notion of a simulation structure is motivated by Weyhrauch's belief that in AI all aspects of a logic should be (at least partly) mechanizable. As implemented here, however, there is the severe problem that the semantically attached code in S need not satisfy the axioms of F, resulting in inconsistency. Note that semantic attachment is quite different from Boyer and Moore's use of

compiled explicit value-preserving functions (see section 6.2) which are proved in (Boyer & Strother Moore, 1981) to satisfy their object-level definitions.

We believe that the notions of simulation structure and semantic attachment are unnecessary. If one wishes to reason about the structures denoted by a language then this should be by formalization in a constructive theory, such as NuPRL, not attachments to an informal programming language. It appears that semantic attachment is not an essential aspect of the reflection mechanism here but merely a substitute for the intrinsic notion of computation which classical logic traditionally lacks. Boyer and Moore's NQTHM (Boyer & Moore, 1979) makes up for this by having a principle of definition whereby the axioms defining a function can be used to give it a computational sense. In NuPRL (Constable *et al*, 1986) computation is an intrinsic notion.

The user is able to direct FOL's attention to any particular LS pair, of which there may be any number. Among these LS pairs Weyhrauch defines a distinguished one called META. This is a theory *of* LS pairs. With an axiomatization of the proof theory of FOL, pre-defined predicates such as TERM, WFF and THEOREM, and semantic attachments to the representations of terms, formulas, etc. used in FOL to implement LS pairs, it serves as the metatheory of an ordinary theory[8]. META is linked to an object theory T via the REFLECT command which embodies the reflection principle

$$\frac{\vdash^{M} \mathrm{THEOREM}(\text{“}w\text{”})}{\vdash^{T} w}$$

This is syntactic reflection. Typically a metatheorem (like (6.2) below) in META is instantiated with constants attached to terms or theorems of T. A constant of META is then attached to the result of evaluating the original term $f(\overline{x})$ of the metatheorem. This should represent a theorem of T which can be "reflected back" using the principle in reverse. Note that although, like Boyer and Moore, reflection is "hard-wired" into FOL, being a general principle it is much less restrictive.

[8]Weyhrauch makes the point that FOL itself is the simulation structure of META.

The reflection process exemplifies Weyhrauch's aim of changing "theorem proving in the theory into evaluation in the metatheory," which he believes explains the traditional procedural/declarative dichotomy. By this is meant the way a (meta)theorem has its usual declarative sense when reasoning in META, and is given a procedural sense by being used to construct proofs via reflection. The other aspect of this, extensibility of reasoning by developing metatheorems rather than the ad hoc addition of routines, is also stressed. FOL is used in a conversational manner and it is unclear to what extent meta-level information is used automatically or how search would be guided. In fact reflection is never used for steps where any search is involved, consisting instead merely of instantiating metatheorems. From the examples presented it would seem that there is no mechanism besides reflection for adding tactics to FOL. Apart from a few built-in decision procedures reasoning takes place at the level of primitive inference rules.

As noted, semantic attachment is again used, in this case to attach representations of proofs etc. in T to constants in META, where we believe a quotation mechanism would be more appropriate. Thus while, like Knoblock, the objective is to provide a uniform language for theory and metatheory, the use of semantic attachment has the effect of pulling the formal system "downwards" towards the implementation level instead of vice versa.

(Weyhrauch, 1980) provides several examples of how a metatheory can be used. One is the example given in chapter 1 where a simple syntactic characterization of the provable formulas of the $\{\leftrightarrow\}$-fragment is given. More generally, metatheorems have the form

$$\forall \overline{x}.P(\overline{x}) \rightarrow \text{THEOREM}(f(\overline{x})) \tag{6.2}$$

where P is what Knoblock has called an applicability predicate, with the same implied "speed-up" benefit, and f is a metafunction constructing the output formula. This is very similar to the form of metatheorem we use.

(Weyhrauch, 1980; Weyhrauch, 1982) provide other examples which use the metatheory to provide features typical of a higher-order language. Examples of these

are derived rules abstracted w.r.t. formulas, and the expression and instantiation of schemas, e.g., for induction.

As with Knoblock, Weyhrauch generalizes the idea of a metatheory to a tower of meta^{i}theories, looking in particular at the applications of a metametatheory. Amplifying the central notion of object-level theorem proving as meta-level evaluation, he suggests this as the proper setting for the expression of a system's heuristics. He describes a small induction heuristic, based on Boyer and Moore's, in this theory. As noted above, there is a blurring of what is strictly "meta" and what is really simulation of higher-order features; it appears that the examples requiring a metametatheory here, in NuPRL require no more than a metatheory.

Also like Knoblock, Weyhrauch considers collapsing the tower of metatheories into one *self-reflective* theory. This is done by making the theory linked to META META itself. Since the general reflection principle described above is retained this leads to a second, fundamental, form of inconsistency. Weyhrauch admits that it is "possible to ask META embarrassing questions."

FOL is inadequate from the aspect of monotonicity in several ways. First, it is not clear how a metatheory is associated with its particular object theory. Nor is it explained how a metatheory is abstracted w.r.t. a theory so that when THEOREM is used, for instance, the object theory to which it refers is clear. Finally, there is no notion of monotonicity itself. As an object theory is enlarged, previously correct metatheorems may becomes invalid. This is yet another point at which inconsistency may enter.

(Weyhrauch, 1980) is more concerned with how metatheorems are used than with proving their correctness. Weyhrauch compares the act of proving a metatheorem, by simulating an object-level proof, to *teaching* FOL. (Weyhrauch, 1982) examines the correctness problem in more detail, giving a step-by-step "teaching" proof of a derived rule. No ideas for the automation of this task in general are expressed, though Weyhrauch talks of coding in a metametatheory a heuristic, reminiscent of example-based generalization, to lift object-level theorems to generic derived rules.

6.5.1 Summary

FOL is attractive for reflection because of its ability to switch easily between theories, and, of importance for syntactic reflection, its relatively simple proof theory. (Weyhrauch, 1980) makes clear the advantages of reflection from both the theorem proving and AI perspectives. However, nowhere are the issues of automation of correctness proofs discussed, nor those of synthesis.

There are important problems with FOL. The three ways in which inconsistency can arise — unsound use of semantic attachment, metatheorems at variance with the object theory, and use of a general self-reflection principle — must be addressed. Semantic attachment should be replaced by a principled notion of computation and a quotation mechanism. Problems of monotonicity are not recognized. The above can be forgiven since (Weyhrauch, 1980) states that its intention is not to deal with these points but to propose a new paradigm for formalising reasoning. Judging by the number of ideas originating in (Weyhrauch, 1980) and their prevalence elsewhere now, it has has been successful in this.

6.6 Davis and Schwartz

(Davis & Schwartz, 1977) provides perhaps the earliest example of computationally-oriented syntactic reflection and is cited as an influence in (Knoblock, 1987). It is shown how new inference rules, similar to the kinds of tactics we use, may be soundly added to a system.

The emphasis in (Davis & Schwartz, 1977) is the opposite of Weyhrauch's. Where the latter was concerned with testing the usefulness of some "epistemological ideas" to AI, the former grew out of work on raising the level of reasoning in program verification systems with stress consequently placed on formal correctness and scalability. A distinction is made with earlier work in this area which applied only to small classes of problems: Davis and Schwartz intend their mechanism to serve as a general framework in which more specific systems can be included. In line

with this aim, a "logical prototype" of a verification system, VT, is defined, and a means of adding new inference rules, and the logical and metamathematical issues raised by this, studied.

The formality is provided by a formal system, FS, based on an axiomatization of set theory in classical first-order predicate calculus[9]. A decidable subset of this language, LFS (hereditarily finite set theory), is identified by banning ω from the language and limiting all quantification to the bounded variety. This allows the definition of a notion of computation ($\rightsquigarrow$) for all sentences of LFS by recursive evaluation of boolean truth values. One of the requirements of FS is that it be complete w.r.t. LFS's evaluation, i.e. that for any sentence ϕ of LFS:

$$\phi \rightsquigarrow \text{true} \iff \vdash_{\text{FS}} \phi \tag{6.3}$$

Thus, LFS provides a formal, though very inefficient, programming language. VT, a system for verifying theorems in FS, is implemented in a "programming environment" initially assumed to be LFS[10]. VT consists of a pair of sets, $< \text{VA}, \text{RI} >$, the verified assertions and rules of inference known to the system. After showing how natural numbers, tuples and lists can be represented in set theory, LFS is used to formalize the various concepts of FS's proof theory analogously to the way primitive recursive arithmetic is used to formalize number theory in (Gödel, 1931). Proofs are represented as linear lists of formulas, and the end result of this Gödelization of FS is the LFS predicate PROVE("ϕ", π) encoding the assertion that π is (the representation of) a proof of the sentence ϕ. The following predicate may be defined in FS but not LFS, and encodes theoremhood:

$$\text{THM}(X) \;=\; \exists Y.\text{PROVE}(X, Y) \tag{6.4}$$

---

[9](Davis & Schwartz, 1977) is not precise in this and it appears that anything subsuming number theory is sufficient.

[10]The FS-LFS relationship is not essential but convenient due to the common language and completeness result (6.3).

Proofhood in VT of course involves extra-logical references to VA and RI. These are encoded in LFS by the predicates:

$$\text{SOME\_VA}(X) = (X = \text{“}\phi_1\text{”}) \vee \cdots \vee (X = \text{“}\phi_m\text{”}) \quad (6.5)$$

$$\text{SOME\_RI}(X) = \Phi_1(X) \vee \cdots \vee \Phi_n(X) \quad (6.6)$$

The rules of inference of RI are formulas, $\Phi(X)$, of LFS having exactly one free variable. The sentence ϕ of FS may be inferred by Φ from the sentences $\phi_1, \ldots, \phi_k$ if $\Phi([\text{“}\phi_1\text{”}, \ldots, \text{“}\phi_k\text{”}, \text{“}\phi\text{”}]) \rightsquigarrow$ true. This, being in LFS and therefore decidable, ensures proof-checking in VT is effective. (It also allows SOME_RI to mention the Φ_i directly rather than encode them as in SOME_VA.) In VT's initial state, VA will be empty and RI will contain rules corresponding to the axioms of FS and the inference rules of the predicate calculus.

Various modes are prescribed in which VT may be used. These include modes for checking proofs and adding definitions of functions and relations, but it is mode 4, "rule insertion", which is of interest to us. To add a rule of inference Φ to RI one must first provide a correctness assertion α of the form

$$\forall X.\Phi(X) \rightarrow (\forall j.1 \leq j < \text{Len}(X) \rightarrow \text{THM}(X(j))) \rightarrow \text{THM}(X(\text{Len}(X))) \quad (6.7)$$

where $\text{Len}(X)$ is the length of the the list X and $X(j)$ is its jth member. As well as α one must provide a proof π which VT checks satisfies $\text{PROVE}(\text{“}\alpha\text{”}, \pi) \rightsquigarrow$ true.

From this it can be seen that VT's inference rules correspond quite closely to Knoblock's partial tactics (see section 6.4) and our synthetic tactics, the applicability predicate being implicit in Φ. The latter complicates "high-level" reasoning slightly. Davis and Schwartz suggest an application where a new inference rule for "natural" proofs in algebra is added but this involves a nested embedding of the encoding of these natural proofs in the existing encoding of proofs and the necessary switching between these.

An important difference between VT and the other systems described here is VT's orientation towards verification (proof-checking) rather than theorem proving. Besides the problem of support in constructing the possibly very large proof terms π which the user must supply to the system, the members of RI can only be used

to check proof steps rather than construct them as traditional tactics do. We don't see this as an essential difference since one could use functions of LFS rather than sentences to represent inference rules and revise the other notions of proof accordingly.

From the point of view of constructivity, (6.7) requires that the user show that proofs of the premises can be used to build a proof of the conclusion. However, since FS is classical and proofs are not stored along with theorems in VA, one does not have the "proof-on-demand" capability found in Knoblock's work.

If we regard computation in LFS as metatheory, then what one has in VT is not separately implemented object- and meta-theories but an object theory (FS) implemented in the metatheory (like (Knoblock, 1987)). Thus, in VT the notation $\vdash_{FS} \phi$ (we shall drop the subscript from now on) means "there is an LFS term π such that PROVE("ϕ", π) $\rightsquigarrow$ true". Reflection in this setting takes the following form. Suppose we have the goal ϕ which follows from the subgoals ϕ_j by rule $\Phi \in$ RI. From $\vdash \phi_j$ we get $\vdash$ THM("ϕ_j") this is the lifting part and justified by the completeness of FS w.r.t. LFS and a set-theoretic property of FS. The completeness of FS w.r.t. LFS also gives us $\vdash \Phi$(["ϕ_1", ..., "ϕ_k", "ϕ"]) from the assumption that Φ(["ϕ_1", ..., "ϕ_k", "ϕ"]) $\rightsquigarrow$ true. This, the lifted subgoals and the correctness theorem (6.7) for Φ give us, assuming the usual predicate calculus rules of inference hold in FS, give us $\vdash$ THM("ϕ"). This step is what has previously been described as "running the metafunction". In VT, with its proof-checking orientation, though Φ may be evaluated to check applicability, the goal ϕ is provided by the user at the outset. The final, dropping, step from $\vdash$ THM("ϕ") to $\vdash \phi$ is justified by the soundness of FS w.r.t. LFS and an assumption by Davis and Schwartz of the weak ω-consistency of FS, a weak constructive property. From this description the distinct properties required of the system FS and of the FS-LFS link are clear. The latter tell us what is required of the programming environment of VT, whether it is implemented in LFS or otherwise.

The lifting and lowering processes described constitute a general principle of self-reflection, viz.

$$\frac{\vdash THM(\text{“}\phi\text{”})}{\vdash \phi}$$

which would seem to make VT inconsistent. Though this can be formalized in VT, it cannot, for theoretical reasons, be proved there.

Davis and Schwartz highlight three properties which they see as essential in a verification system: soundness, extensibility and stability. By extensibility they mean the addition of new rules of inference as described above, and by stability the preservation of soundness across such extensions. In fact, the above justification for reflection applies only in the case that VT doesn't change state, i.e. VA and RI do not vary. Of course, any operation apart from proof-checking *does* change VT's state. (Davis & Schwartz, 1977) proves the stability of VT in a metatheorem similar to that in (Boyer & Strother Moore, 1981): by indexing the successive states of VT with a counter, t, and relativizing the justification of reflection w.r.t. to this. E.g. one has $\text{SOME\_VA}_t(X)$, $\text{PROVE}_t(\alpha, \pi)$, $\text{THM}_t(\alpha)$, etc. By induction on t it is shown that for all t $\vdash_t \phi$ (i.e. there is a π s.t. $\text{PROVE}_t(\text{“}\phi\text{”}, \pi) \rightsquigarrow \mathit{true}$) implies $\vdash_0 \phi$ (previously just $\vdash \phi$). This says that any actions in VT lead only to conservative extensions of its initial state.

The use of an implicit environment, in the form of the current theory as captured by the predicates $\text{SOME\_VA}(X)$ and $\text{SOME\_RI}(X)$, is very much like Boyer and Moore's system (see section 6.2) and has the same advantages and disadvantages. These predicates, of course, change as the state of VT changes, and this poses some problems of non-monotonicity. Their exact status is not made clear in (Davis & Schwartz, 1977). Inclusion as definitions in the normal sense in VT leads to clearly non-conservative behaviour. Some "guarding mechanism", as employed by Boyer and Moore who use meta-axioms and meta-definitions, or similar is required.

Unlike the other systems mentioned in this chapter, Davis and Schwartz give an outline for a quantitative measure of the "speed-up" provided by a reflection mechanism. This is in terms of the reduction of the *difficulty* of a theorem, defined as the length of the shortest input which will cause VT to accept it. It is conjectured that "the availability of these extension principles will reduce the difficulty of large classes of sentences by very large amounts." Although no formal demonstration of

this conjecture is provided, Davis and Schwartz point to some empirical evidence including the algebra rule mentioned above.

The clearest indication of the non-practicability of VT is the way set theory is used as a programming language; this is used to show the theoretical possibility of doing something rather than provide an efficient means. To meet this criticism (Davis & Schwartz, 1977) suggests how a more directly executable programming language might be bootstrapped into the programming environment. This is to proceed by formalizing an abstract machine in LFS, then exhibiting a program for evaluating a subset of LFS which is proved correct w.r.t. the formalization. This, presumably, may then take the place of native LFS evaluation for this subset. Note the resemblance of this to the use of compiled metafunctions in (Boyer & Strother Moore, 1981). It is envisaged that more extensive subsets of LFS will be handled by successive programs, creating a bootstrap effect.

6.6.1 Summary

(Davis & Schwartz, 1977) clearly presents itself as a theoretically-oriented suggestion for implementing extensibility. In particular, its use of set theory to elegantly unify formal system and programming language militate against practical use. Despite this, and though the system, VT, it discusses is a proof *checker*, many of the ideas have appeared in later, practical, work. It's use of applicability predicates has, indirectly, influenced our work. There is no mention of methods for proving the correctness of new rules apart from the implicit bootstrapping provided by previously added rules.

6.7 Higher-order patterns

For reasons of self-containment we briefly describe the concepts and terminology of (Miller, 1991) and (Liang, 1992) in relation to our section 4.8. The algorithm we have used is based on that in (Nipkow, 1991) and listed in appendix F.

Higher-order patterns (h.o.p.'s) are a subset of terms of the λ-calculus. A λ-term t in β-normal form is a h.o.p. if: every free occurrence of a variable F in t is in a subterm $(F\ a_1 \ldots a_n)$ where a_i are η-equivalent to distinct bound variables.

The restriction on free variables in h.o.p.'s means that as they become instantiated only a very simple form of β-reduction renaming or β_0-reduction is possible. This results in a greatly simplified unification problem for h.o.p.'s. Miller shows that h.o.p. unification is decidable and yields a most general unifier. (Qian, 1992) improves this by providing an algorithm linear in time and space. In contrast, general unification of simply-typed λ-terms of order 2 and higher is undecidable.

The h.o.p. unification algorithm makes no use of types. As we have used it in section 4.8 our terms could be simply-typed at the level of syntax, i.e. saturation and arities, but this is not done.

(Liang, 1992) showed how h.o.p.'s could be used to annotate terms with wave fronts and holes by including a `wave` function at the object level. For example, the annotated term $\boxed{f(\underline{x})}$ would become the term (`wave` $f\ x$). This approach has the drawbacks of moving meta-level annotation to the object level and, as a consequence, requiring `wave` to be polymorphically typed. We have instead used `wave` as a meta-level annotation of OYSTER's abstract syntax where neither problem exists. We have shown how to express potential wave-fronts in h.o.p.'s by placing higher-order variables in wave holes.

Our work necessitated the use of some form of annotation for higher-order terms, and in practice it was found that h.o.p.'s sufficed. Unfortunately, this is not the case for proof planning in general, for example (Madden *et al*, 1993) where a less tractable class of unification is required.

6.8 Conclusions

The main difference we see between previous work and ours is the kind of tactics we are interested in, and, more importantly, that we are attempting to (at least partially) automate the process of their synthesis. Howe provides tactics which are able to take object-level equational theorems and produce corresponding meta-level rewrites. He also gives powerful type-checking tactics which are usually sufficient to prove the correctness of rewrites built using Paulson style tacticals. In contrast, we are interested in synthesizing tactics from their meta-level specifications. These different objectives result in different mechanisms. For example, to enable sophisticated type checking Howe makes such things as monotonicity an intrinsic part of many definitions, whereas this would greatly complicate the presentation one would give to Clam. By using Clam to plan synthesis proofs we do not need such object-level guidance and are willing to accept the one-shot expense of clumsier proofs.

Most of the work in Clam has concentrated on using the wave annotation to guide methods. Hence, in attempting to synthesize corresponding tactics, we extend the usual notion of term to include this extra annotation and are principally interested in tactics which make use of or manipulate such annotation. Neither of these aspects, meta-annotation and synthesis, is explored in the work cited.

Chapter 7

Conclusions and Further Work

7.1 Recap

We were originally motivated to use the *CLAM* proof planner and the proof plan paradigm to attempt to automate, at least partly, the task of synthesizing tactics to fit *CLAM* method specifications. The wider ramifications of success in this programme were laid out in chapter 1.

We feel that this goal has to a great extent been achieved. In order to do this we had to make several design choices:

It was first necessary to decide how we could capture the properties of a tactic with sufficient precision so that the specification given by a method made sense. Treating tactics as simply a special class of program this entailed choosing one suitable program specification formalism, from among the many available, which was best suited to this purpose.

We chose to use the constructive type theory OYSTER. But even after this choice the behaviour of tactics seemed complicated beyond the reach of present-day program synthesis techniques.

To make the problem tractable we narrowed down what it means to be a tactic, taking note of the existing systems (Howe, 1988a; Knoblock, 1987) and

(Boyer & Strother Moore, 1981), and arriving at the notion of a tactic as a partial rewrite function.

Since tactics must do more than mere syntax manipulation we required a correctness criterion to ensure no unsoundness could arise. Such a criterion is provided by a reflection mechanism. A simplified version of Howe's mechanism was successfully implemented in OYSTER.

Howe's full mechanism would have introduced too much unnecessary technical obfuscation into synthesis proofs. It was instead decided to restrict our attention to rewriting a small subset of terms of type `pnat`.

By simulating the predicates appearing in methods as OYSTERobject-level predicates the necessary amount of representation of $CLAM$ at the meta-level was in place to allow synthesis problems to be tackled.

We then had to decide what amounted to a synthesis in this formalism. Following the example of (Knoblock & Constable, 1986) our correctness theorems took the form:

$$\forall i : \texttt{metaterm.preconds}(i) \rightarrow \exists o : \texttt{metaterm.effects}(i, o) \wedge i \sim o$$

where `preconds`' truth entailed the success of the tactic. This semantics is almost identical to the informal semantics of the $CLAM$ methods themselves.

We also stipulate that the `preconds` should be decidable where possible. This gives us useful properties for the extract and is essential to compiling a pseudo tactic.

After conducting several example proofs it became clear that $CLAM$ lacked some of the machinery necessary to enable a successful proof. Principal among these we context-sensitive rewriting and higher-order rippling. We found existing recursion analysis only required slight modifications to make it sufficient for our needs.

A means of compiling Prolog programs from decidability proofs was developed. This is not presently well developed but promises the solution to the original problem: tactics which are guaranteed to work.

7.2 Further work

Several clear problems arose during the course of this work. We list them here with suggested directions for further work.

More general tactics

As pointed out in several places, the form of tactic we synthesize in this thesis is of a very limited nature. It is limited to rewriting subterms of type pnat. (Howe, 1988a) and (Knoblock, 1987) suggest two possible solutions of which the former seems more immediately realistic. We would, if we could reflect sequents and objects of higher type, be able to synthesize a much more useful class of tactic. These include weak and strong fertilization, first-order rewrites of general type, and wave. There would at last be some real bootstrapping of CL^AM tactics.

Efficiency

The means of tactic specification throughout this thesis was intended to be perspicuous rather than efficient. Methods are known to eliminate the unwanted non-computational components from proof extracts, e.g. subset types in place of existential types. It would be interesting to see how far the ideas in this thesis could be redone using the subset type.

Another means to improve efficiency would be the addition of a primitive `metaterm` type to OYSTER. This would allow cleaner extracts and more efficient recursion. Alternatively Nordström's `acc` type could be used (Nordström, 1988).

Compound tactics

We have chosen only to verify compound tactics. Large specifications must still be synthesized from the ground up as an atomic tactic. This would seem to put low limit on complexity of tactic we can synthesize. A means of decomposing large tactics into simpler synthesis problems and composing the results holds more promise as methods grown in complexity.

Pseudo tactics

There is a large amount of room for improvement in the process of compilation to Prolog. Anti-floundering ordering of literals, mode specific programs and elimination of unnecessary term structure will lead to better proof planning.

7.2.1 A kernel meta-language

Examination of the literature on program specification (see chapter 2) showed the clear development of a school of thought which believes that much can be gained by simplifying the specification language as far as possible. The idea, exemplified in (Sannella & Wirsing, 1983), is that a kernel language containing the bare essentials for specification should be given, and that this should be made palatable to the human user by constructing a higher-level interface language in terms of the kernel. The latter could come in different "flavours" according to its intended use.

We believe that a similar approach would be fruitful in designing a meta-language for *CLAM*. A large number of predicates have been accumulated during its lifetime, many differing only slightly in function. Rationalization of these in terms of a small kernel language would make the system more uniform and comprehensible to the user, as well as providing the usual benefits of modular software. The kernel language could, for example, be based on an abstract and unified data type for annotated terms, and contain a small number of syntax property and "surgery" operations. It would also be interesting to re-examine the work of this thesis using

such a kernel specification language instead of representing each CL^AM predicate individually. Research in this direction has begun with an abstract definition of annotated terms.

7.2.2 A quotation type

As noted by Howe (see section 6.3), and experienced also with our reflection mechanism, the process of lifting and lowering consumes large amounts of time and space when metaterms are formalized in the usual way in type theory. Boyer and Moore (see section 6.2), however, show that an efficient implementation is possible by including a purposely designed quoting structure at the implementation level. Attempts have already been made in this direction ((Allen, 1990; Knoblock & Constable, 1986)) but these are based on syntactic reflection and are somewhat unwieldy. We believe that the introduction of a type analogous to Boyer and Moore's use of the `QUOTE` would have the following benefits. First, a large part of the library necessary for Howe's mechanism would be obviated, allowing metafunctions to be developed more easily and with less overhead. Second, reflection would be made time and space efficient and could thus be used as an integral part of the system, as in NQTHM.

As a type theoretic reconstruction of Boyer and Moore's implementation we suggest a type, $\texttt{quotation}(A)$, having the following rules:

$$\frac{H >> A \texttt{ in U}i}{H >> \texttt{quotation}(A) \texttt{ in U1}} \text{ intro at Ui}$$

$$\frac{H >> t \texttt{ in } A}{H >> \texttt{quote}(t) \texttt{ in quotation}(A)} \text{ intro}$$

$$\frac{????}{H, x : \texttt{quotation}(A), H' >> G} \text{ elim x}$$

$$\frac{H >> t \texttt{ in } A}{H >> \texttt{dequote}(\texttt{quote}(t)) = t \texttt{ in } A???????} \text{ reduce}$$

7.3 Summary

We have shown that simple tactic synthesis is both possible and well with in the capabilities of the *CLAM* theorem prover. We have made this possible by extending the rewriting ability of *CLAM* and its notion of wave rule. We have shown that the *CLAM* proof plan is also suited to carrying out decidability proofs and have demonstrated a primitive means of compiling the results to faster pseudo-tactics.

References

? (1992). ? Paper submitted to CADE-11.

Aiello, L. and Weyhrauch, R.W. (1980). Using meta-theoretic reasoning to do algebra. In Bibel, W. and Kowalski, R., (eds.), *5th Conference on Automated Deduction*, pages 1 13. Springer Verlag. Lecture Notes in Computer Science No. 87.

Allen, Stuart F. (1986). *The semantics of type theoretic languages*. Unpublished Ph.D. thesis, Dept of Computer Science, Cornell Universtiy.

Allen, S. F. (1990). The semantics of reflected proof. Research Report ??, Cornell University.

Apt, K. R. (1978). A sound and complete Hoare-like system for a fragment of Pascal. Technical Report IW 97/78, Amsterdam Mathematisch Centrum.

Bishop, E. (1967). *Foundations of Constructive Analysis*. McGraw-Hill, New York.

Bowen, J. P. and Gordon, M. J. C. (1994). Z and HOL. In *Z user workshop, Cambridge 1994*. Workshops in Computing. Springer-Verlag. To appear.

Boyer, R.S. and Moore, J.S. (1979). *A Computational Logic*. Academic Press, ACM monograph series.

Boyer, R.S. and Strother Moore, J. (1981). Metafunctions. In Boyer, R.S. and Strother Moore, J., (eds.), *The Correctness Problem in Computer Science*, pages 103 184. Academic Press.

Brown, F.M. (March 1977). The theory of meaning. Research Report 35, Dept of Artificial Intelligence, Edinburgh.

Bundy, A., van Harmelen, F., Hesketh, J. and Smaill, A. (1988). Experiments with proof plans for induction. Research Paper 413, Dept. of Artificial Intelligence, Edinburgh, To appear in JAR.

Bundy, A., Smaill, A. and Hesketh, J. (1989a). Turning eureka steps into calculations in automatic program synthesis. Research Paper 448, Dept. of Artificial Intelligence, Edinburgh, In proceedings of UK IT 90.

Bundy, A., van Harmelen, F., Hesketh, J., Smaill, A. and Stevens, A. (1989b). A rational reconstruction and extension of recursion analysis. In Sridharan, N.S., (ed.), *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, pages 359 365. Morgan Kaufmann. Also available from Edinburgh as DAI Research Paper 419.

Bundy, A., Smaill, A. and Wiggins, G. A. (1990a). The synthesis of logic programs from inductive proofs. In Lloyd, J., (ed.), *Computational Logic*, pages 135 149. Springer-Verlag, Esprit Basic Research Series. Also available from Edinburgh as DAI Research Paper 501.

Bundy, A., van Harmelen, F., Smaill, A. and Ireland, A. (1990b). Extensions to the rippling-out tactic for guiding inductive proofs. In Stickel, M.E., (ed.), *10th International Conference on Automated Deduction*, pages 132 146. Springer-Verlag, Lecture Notes in Artificial Intelligence No. 449. Also available from Edinburgh as DAI Research Paper 459.

Bundy, A., Stevens, A., van Harmelen, F., Ireland, A. and Smaill, A. (1991). Rippling: A heuristic for guiding inductive proofs. Research Paper 567, Dept. of Artificial Intelligence, Edinburgh, In the Journal of Artificial Intelligence.

Bundy, A., Stevens, A., van Harmelen, F., Ireland, A. and Smaill, A. (1993). Rippling: A heuristic for guiding inductive proofs. *Artificial Intelligence*, 62:185 253. Also available from Edinburgh as DAI Research Paper No. 567.

Burstall, R.M. and Darlington, J. (1977). A transformation system for developing recursive programs. *Journal of the Association for Computing Machinery*, 24(1):44 67.

Burstall, R.M. and Goguen, J.A. (1977). Putting Theories Together to Make Specifications. In *Proceedings of the 5th IJCAI*, pages 1045 1058.

Burstall, R. and McKinna, J. (1991). Deliverables: an approach to program development in the calculus of constructions. LFCS report 91 133, Laboratory for Foundations of Computer Science, Edinburgh.

Burstall, R. M. (1974). Program proving as hand simulation with a little induction. In Rosenfeld, J. L., (ed.), *Information Processing '74*.

Chadha, R. and Plaisted, D. A. (1993). On the mechanical derivation of loop invariants. *Journal of Symbolic Computation*, 15:705 744.

Chang, C-L. and Lee, R. C-T. (1973). *Symbolic logic and mechanical theorem proving*. Academic Press.

Church, A. (1951). Special cases of the decision problem. *Revue philosophique de Louvain*, 49:203 221. A correction, ibid., vol. 50 (1952), pp. 270 272.

Clark, K.L. (1978). Negation as failure. In Gallaire, H. and Minker, J., (eds.), *Logic and Data Bases*, pages 293 322. Plenum Press.

Clarke, E. M. C. (1979). Programming language constructs for which it is impossible to obtain good Hoare axiom systems. *Journal of the Association for Computing Machinery*, 26:129 147.

Constable, R.L. and Howe, D.J. (1990). Implementing metamathematics as an approach to automatic theorem proving. In Banerji, R.B., (ed.), *Formal Techniques in Artificial Intelligence*, pages 45 76, Amsterdam. North Holland.

Constable, R.L., Allen, S.F., Bromley, H.M. *et al.* (1986). *Implementing Mathematics with the Nuprl Proof Development System*. Prentice Hall.

Cook, S. A. (February 1978). Soundness and completeness of an axiom system for program verification. *SIAM Journal on Computing*, 7(1):70 90.

Coquand, Th. and Huet, G. (1988). The calculus of constructions. *Information and Computation*, 76:95 120.

Davis, M. and Schwartz, J. (1977). Metamathematical extensibility for theorem verifiers and proof checkers. Technical Report 12, Courant Institute of Mathematical Sciences, New York.

de Bakker, J. (1980). *Mathematical Theory of Program Correctness*. Prentice-Hall.

Ehrig, H. and Mahr, B. (1985). *Fundamentals of algebraic specification 1*. Springer-Verlag.

Floyd, R. W. (1967). Assigning meanings to programs. In Society, American Mathematical, (ed.), *Mathematical aspects of computer science*, pages 19 32.

Futatsugi, K., Goguen, J.A., Jouannaud, J-P. and Meseguer, J. (1985). Principles of OBJ2. In Reid, B., (ed.), *Proceedings, Twelfth ACM Symposium on Principles of Programming Languages*, pages 52 66. Association for Computing Machinery.

Girard, J.-Y. (1987). *Proof Theory and Logical Complexity*, volume 1. Bibliopolis, Naples.

Giunchiglia, F. and Traverso, P. (1990). Plan formation and execution in a uniform architecture of declarative metatheories. Technical Report 9003-12, Instituto per la Ricerca Scientifica e Tecnologia, Trento, Italy.

Goad, C. A. (1980). *Computational uses of the manipulation of formal proofs*. Unpublished Ph.D. thesis, Stanford University.

Gödel, K. (1931). Über formal unentscheidbare sätze der principia mathematica und verwandter systeme i. *Monatsh. Math. Phys.*, 38:173 98. English translation in (Heijenoort, 1967).

Goguen, J. and Winkler, T. (1988). Introducing OBJ3. Technical Report SRI-CSL-88-9, SRI International, Computer Science Lab.

Goguen, J. A. (1986). More thoughts on specification and verification. In Gehani, N. and McGettrick, A. D., (eds.), *Software specification techniques*, pages 47 52. Addison-Wesley.

Goguen, J.A. (1989). OBJ as a theorem prover, with application to hardware verification. In Subramanyan, V.P. and Birtwhistle, G., (eds.), *Current Trends in Hardware Verification and Automated Theorem Proving*, pages 218 267. Springer. Also Technical Report SRI-CSL-88-4R2, SRI International, Computer Science Lab, August 1988.

Goguen, J. A., Thatcher, J. W. and Wagner, E. G. (1978). An initial algebra approach to the specification, correctness and implementation of abstract data types. In Yeh, R., (ed.), *Current trends in programming methodology IV: data structuring*, pages 80 144. Prentice-Hall.

Gordon, M. J. C. (1988). *Programming Language Theory and its Implementation*. Prentice-Hall.

Gordon, M., Milner, R., Morris, L., Newey, M. and Wadsworth, C. (September 1977). A metalanguage for interactive proof in LCF. Internal report CSR-16-77, Dept of Computer Science, University of Edinburgh.

Gordon, M., Milner, R. and Wadsworth, C. (September 1977). Edinburgh LCF. Internal report CSR-11-77, Dept of Computer Science, University of Edinburgh.

Gordon, M.J., Milner, A.J. and Wadsworth, C.P. (1979). *Edinburgh LCF - A mechanised logic of computation*, volume 78 of *Lecture Notes in Computer Science*. Springer Verlag.

Hamilton, A. (1993). Automatic generation of theories. Technical Report 100, University of Stirling.

Harel, D. (1980). Proving the correctness of regular deterministic programs: a unifying survey using dynamic logic. *Theoretical Computer Science*, 12:61 81.

Harel, D. (1984). Dynamic logic. In *Handbook of Philosophical Logic, Vol. II*, pages 497 604. D. Reidel Publishing Co.

Harel, D. (1988). *Algorithmics*. Addison-Wesley.

Heyting, A. (1956). *Intuitionism: an introduction*. North-Holland.

Hindley, J. R. and Seldin, J. P. (1986). *Introduction to combinators and λ-calculus*. Cambridge University Press.

Hoare, C. A. R. and Wirth, N. (1973). An axiomatic definition of the programming language Pascal. *Acta Informatica*, 2:335 355.

Hoare, C. A. R. (1969). An axiomatic basis for computer programming. *Communications of the ACM*, 12:576 58.

Horn, C. (1988). The Nurprl proof development system. Working paper 214, Dept. of Artificial Intelligence, Edinburgh, The Edinburgh version of Nurprl has been renamed Oyster.

Howard, W.A. (1980). The formulae-as-types notion of construction. In Seldin, J.P. and Hindley, J.R., (eds.), *To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 479 490. Academic Press.

Howe, D. J. (1988a). *Automating Reasoning in an Implementation of Constructive Type Theory*. Unpublished Ph.D. thesis, Dept of Computer Science, Cornell Universtiy.

Howe, D.J. (1988b). Computational metatheory in Nuprl. In Lusk, R. and Overbeek, R., (eds.), *9th Conference on Automated Deduction*, pages 238 257. Springer-Verlag.

Howe, D. (1989). Equality in lazy computation systems. In *LICS '89*.

Huet, G. (August 1977). Confluent reductions: Abstract properties and applications to term rewriting systems. Rapport de Recherche 250, Laboratoire de Recherche en Informatique et Automatique, IRIA, France.

Ireland, A. and Bundy, A. (1992). Using failure to guide inductive proof. Technical report, Dept. of Artificial Intelligence, Edinburgh, Available from Edinburgh as DAI Research Paper 613.

Ireland, A. (1992). The Use of Planning Critics in Mechanizing Inductive Proofs. In Voronkov, A., (ed.), *International Conference on Logic Programming and Automated Reasoning LPAR 92, St. Petersburg*, Lecture Notes in Artificial Intelligence No. 624, pages 178 189. Springer-Verlag. Also available from Edinburgh as DAI Research Paper 592.

J.-Y. Girard, Y. Lafont and Taylor, P. (1989). *Proofs and Types*. Cambridge University Press.

Jones, R. B. (Winter 1992). ICL ProofPower. *BCS FACS FACTS*, Series III, 1(1):10 13.

Knoblock, T. B. and Constable, R.L. (1986). Formalized metareasoning in type theory. In *Proceedings of LICS*, pages 237 248. IEEE.

Knoblock, T. (1987). *Metamathematical extensibility in type theory*. Unpublished Ph.D. thesis, Dept of Computer Science, Cornell Universtiy.

Kolmogorov, A. (1932). Zur deutung der intuitionistischen logik. *Matematische Zeitschrift*, 35:58 65.

Kreitz, C. (1993). Meta-synthesis: deriving programs that develop proofs. Technical report, Technische Hochschule Darmstadt.

Liang, C. (1992). Lambda-prolog implementation of ripple rewriting. Abstract, Carnegie-Mellon University.

Lloyd, J.W. (1987). *Foundations of Logic Programming.* Symbolic Computation. Springer-Verlag, Second, extended edition.

Loveland, D.W. (1978). *Automated theorem proving: A logical basis*, volume 6 of *Fundamental studies in Computer Science*. North Holland.

Madden, P. (1991). *Automated Program Transformation Through Proof Transformation.* Unpublished Ph.D. thesis, University of Edinburgh.

Madden, Peter, Hesketh, Jane, Green, Ian and Bundy, Alan. (1993). A general technique for automatically optimizing programs through the use of proof plans. Research Paper 608, Dept. of Artificial Intelligence, Edinburgh, Extended abstract to appear in Proceedings of LOPSTR-93.

Manna, Z. and Waldinger, R. (1978). Is sometime 'sometimes' better than 'always'? *Communications of the ACM*, 21.

Martin-Löf, Per. (August 1979). Constructive mathematics and computer programming. In *6th International Congress for Logic, Methodology and Philosophy of Science*, pages 153 175. Hanover. Published by North Holland, Amsterdam, 1982.

Martin-Löf, Per. (1984). *Intuitionistic Type Theory.* Bibliopolis, Naples, Notes by Giovanni Sambin of a series of lectures given in Padua, June 1980.

Mendler, N. (1986). *Recursive types and infinite objects.* Unpublished Ph.D. thesis, Dept of Computer Science, Cornell Universtiy.

Miller, D. (1991). A logic programming language with lambda abstraction, function variables and simple unification. In *Extensions of Logic Programming*, volume 475 of *Lecture Notes in Artificial Intelligence*. Springer-Verlag.

Negrete, S. (March 1992). Hint Mechanism for Clam. Technical Paper 8, Dept. of Artificial Intelligence, Edinburgh.

Nipkow, T. (1991). Higher-order critical pairs. In *Proc. 6th IEEE Symp. Logic in Computer Science*, pages 342 349.

Nordström, K. B. Petersson and Smith, J. (1990). *Programming in Martin-Löf Type Theory*. Oxford University Press.

Nordström, B.S. (October 1988). Terminating general recursion. *BIT*, 28(3):605 19.

O'Donnell, M. J. (1982). A critique of the foundations of Hoare-style programming logics. *Communications of the ACM*, 25:927 935.

Paulin-Mohring, C. (1989). Extracting F_ω's programs from proofs in the calculus of constructions. *ACM Proc. POPL*.

Paulson, L. (1983a). A higher-order implementation of rewriting. *Science of Computer Programming*, 3:119 149.

Paulson, L. (1983b). *Logic and Computation: interactive proof with Cambridge LCF*. Cambridge University Press.

Qian, Z. (1992). Unification of higher-order patterns in linear time and space. Technical Report 5/92, FB 3 Informatik, Universität Bremen.

Sannella, D. and Wirsing, M. (1983). A kernel language for algebraic specification and implementation. Report CSR-131-83, Dept of Computer Science, Edinburgh.

Sannella, D. (July 1988). A survey of formal software development methods. Report ECS-LFCS-88-56, Laboratory for Foundations of Computer Science, Edinburgh.

Smith, J. (1993). An interpretation of Kleene's slash in type theory. In *Logical Environments*, pages 189 197. Cambridge University Press.

Spivey, J. M. (1988a). *Understanding Z*. Cambridge University Press.

Spivey, J. M. (1988b). *The Z notation*. Prentice-Hall.

Stevens, A. (1989). *An improved method for the mechanisation of inductive proof*. Unpublished Ph.D. thesis, Dept of Artificial Intelligence, Edinburgh.

Sundholm, G. (1983). Constructions, proofs and the meanings of logical constants. *Journal of Philosophical Logic*, 12:151 172.

Turing, A. M. (1936). On computable numbers, with an application to the Entscheidungsproblem. *Proc. London Math. Soc.*, 42:230 265.

van Harmelen, F. (1989). *On the Efficiency of Meta-level Inference*. Unpublished Ph.D. thesis, University of Edinburgh.

van Harmelen, F., Ireland, A., Stevens, A., Negrete, S. and Green, I. (1993). The CLAM proof planner, user manual and programmer manual *(version 2.2)*. to appear as a dai technical paper, DAI.

Wadler, P. (1990). Deforestation: transforming programs to eliminate trees. *Theoretical Computer Science*, 73:231 248.

Weyhrauch, R.W. (1980). Prolegomena to a theory of mechanized formal reasoning. *Artificial Intelligence*, 13:133 170.

Weyhrauch, R.W. (1982). An example of FOL using metatheory: formalizing reasoning systems and introducing derived inference rules. In Loveland, D.W., (ed.), *6th Conference on Automated Deduction*, pages 151 158. Springer-Verlag.

Wiggins, G. A. (1992). Synthesis and transformation of logic programs in the Whelk proof development system. In Apt, K. R., (ed.), *Proceedings of JICSLP-92*, pages 351 368. M.I.T. Press, Cambridge, MA.

Wirsing, M. (1990). *Algebraic specification*, pages 675 788. Elsevier.

Wos, L., Overbeek, R., Lusk, E. and Boyle, J. (1984). *Automated Reasoning: Introduction and Applications*. Prentice-Hall.

Yoshida, Tetsuya. (1993). Coloured rippling. Unpublished M.Sc. thesis, Dept. of Artificial Intelligence, Edinburgh.

Yoshida, Tetsuya, Bundy, Alan, Green, Ian, qWalsh, Toby and Basin, David. (1994). Coloured rippling: An extension of a theorem proving heuristic. Technical Report TBA, Dept. of Artificial Intelligence, Edinburgh.

Appendix A

Oyster and Clam

A.1 The OYSTER theorem prover

When we wish to emphasize that the text we are discussing is computer-generated we will use the `typewriter` font. OYSTER proofs are displayed as in figure A 1. This has several parts. "`example`" is the name of the proof. The remaining pieces of the first line indicate respectively that `[1]` is the position of this node in the proof tree, the proof is partial, and that the current autotactic is `idtac`. The next two lines display the goal sequent for the node: a numbered list of hypotheses (in this case only 1), the OYSTER turnstile "`==>`", and the conclusion. Line 4 indicates the inference rule used to refine the node, and the remainder displays the two resulting subgoals.

The correspondence between OYSTER's syntax and the traditional symbols of mathematics are given in table A 1. For the definitive account of OYSTER see (Horn, 1988).

```
example: [1] partial autotactic(idtac)
1. x:pnat
==> y:pnat=>z:pnat=>x+ (y+z)=x+y+z in pnat
by elim(x)

  [1] incomplete
  ==> y:pnat=>z:pnat=>0+ (y+z)=0+y+z in pnat

  [2] incomplete
  2. v0:pnat
  3. v1:(y:pnat=>
          z:pnat=>v0+ (y+z)=v0+y+z in pnat)
  ==> y:pnat=>
      z:pnat=>s(v0)+ (y+z)=s(v0)+y+z in pnat
```

Figure A–1: Oyster proof format

OYSTER	traditional	description
`pnat`	$\mathbf{N}$	Peano natural numbers
`atom`		character strings
`T list`		lists over T
`s=t in T`	$=$	equality of s and t in type T
`void`	$\bot$	absurdity/empty type
`P # Q`	$P \wedge Q$	conjunction/product
`P \ Q`	$P \vee Q$	disjunction/disjoint sum
`P => Q`	$P \rightarrow Q$	implication/function space
`x:T#P`	$\exists x : T.P$	existential quantification/dependent product
`x:T=>P`	$\forall x : T.P$	universal quantification/dependent function space
`rec(z,T)`		simple recursive type
`u(i)`		universe of level i
`H==>G`	$\mathcal{H} \vdash \mathcal{G}$	(object-level) sequent turnstile

Table A–1:

A.2 The *CLAM* proof planner

In this section we give an overview of the *CLAM* proof planning system (vanHarmelen *et al*, 1993), the nature of the tactics we intend to synthesize and an idea of how we can apply the results in a theorem proving environment. This seems an appropriate point at which to do so since the exercise has a distinct circularity about it: the objects we will synthesize, tactics, are part of the system, *CLAM*, we use to perform the synthesis. This results in the "bootstrapping" capability of the *CLAM* system, another aspect of meta-level reasoning.

A.2.1 Proof planning

At any point in a proof the choice of which inference rule to apply next may be very large, possibly infinite. Since interesting theorems typically require many separate proof steps for completion, this problem of *combinatorial explosion* makes it impossible in practice to successfully employ a brute-force strategy when searching for a proof. Present-day automatic theorem provers attempt to circumvent this problem in various ways. One group, which can be loosely termed *uniform proof procedures*, retain the search for proofs but using a small number of very primitive rules. The resulting search space is further restricted by limiting the applicability of these rules, typically with simple syntactic conditions. Examples of such systems are the resolution theorem provers making use of search restrictions such as SLD, set-of-support, model elimination, locking, and special rules such as paramodulation to handle the explosive use of equality reasoning (Chang & Lee, 1973; Loveland, 1978; Wos *et al*, 1984). The intention is that these systems are complete, and the above search strategies are justified (on paper) by meta-level results showing this. Note, however, that this use of the meta-level is hard-wired into the system, in the form of a fixed proof procedure, and gives the user no opportunity to specialise or improve the system for use in a particular domain.

A second group of theorem provers, which might loosely be termed *tactic-based*, aims to overcome this problem. Proofs, as always, are justified ultimately by primitive inference rules, but the user rarely uses the system at this level.

Just as humans usually carry out proofs by first "strategically" breaking a problem up into conceptually manageable pieces, then solving each of these with well-honed and more specialised "tactical" reasoning methods, so tactic-based systems allow users to write programs *tactics* which are suited to solving small, specific, high-level tasks.

Tactics are intended to be used in a modular fashion, like a library of subroutines is used in writing a program. The analogue of the programming language construct is the *tactical*. Tacticals provide the modularity by allowing one to fit tactics together in a well-understood way to form super-tactics. [See section 2.2 for details of tactic implementation.] In this way, one can build ever higher-level tactics, capable, for example, of implementing the uniform proof procedures described above. Obviously, with this burgeoning of new possibilities for inference, brute-force search is even more hopeless than previously. Thus, tactic-based systems are invariably interactive and rely on an external source to supply a *strategy* able to suitably compose tactics to perform a complete proof. This source can either be human or another computer program. *CLAM* is an example of the latter.

CLAM is a proof planner. Whilst the concept is quite general, the version used here sits on top of the OYSTER proof system, a Prolog re-implementation of NuPRL (Horn, 1988; Constable *et al*, 1986). *CLAM* employs a user-defined list of high-level tactics, putting them together to form a plan to putatively perform a complete proof. In order to guide its search for such a plan the user supplies tactics in the form of *methods* tactics plus an accompanying meta-level specification. *CLAM* searches for a proof purely at the meta-level using these specifications and returns the result of composing the appropriate tactics. The efficacy of this approach has been demonstrated in (van Harmelen, 1989; Bundy *et al*, 1988). The current *CLAM* distribution has a library of methods and submethods (methods which may be called by other methods but are not themselves used at the top level) suited to performing inductive proofs in the OYSTER system, particularly those relating to

the verification and synthesis of programs. A high-level "story", `ind_strat`, can be told for induction proofs. A simplified version of this, where only induction schemas with a single base and step case are considered, has the form:

```
induction(Scheme,IndVar:Type) THEN
        [base_case,
         step_case
        ]
```

Such a unifying plan outline or strategy is called a *proof plan*. One can see from this example that just as tacticals allow one to structure the building of complex tactics, so complex methods can be built using *methodicals*. `THEN` and the list constructor are used to form the top-level `ind_strat` from the `induction` method, which analyses the goal and suggests a suitable induction schema, and the methods `base_case` and `step_case` (see below) which are specialised for their respective subgoals. It is a proof plan for synthesizing tactics that is the objective of this work.

As will be seen in the sequel, induction plays an important part in program synthesis since it is the proof rule which introduces, via the Curry-Howard isomorphism (q.v.), recursion in the extracted program: there is an exact mapping (duality) between the induction schema we use and the form of recursion in the resulting program. Induction is the sole means we have of reasoning, in a non-general way, about the infinite sets of objects (such as numbers, lists and terms) we use as data in programs. Resolution theorem provers, suited to pure first-order predicate logic, are notoriously poor at performing inductive proofs. They are hindered by the impossibility of expressing induction schemas in first-order logic, the loss of structure which results from putting theory and conjecture into clausal form, and the unsuitability of the strategies typically employed to guide search. The latter are usually global in nature, applying equally to all clauses of a theory and, as mentioned above, uniform. It is this low-level uniformity of proof rule, where the same guidelines are applied during disparate parts of a proof, and the consequent inability to make use of higher-level strategic knowledge, which makes resolution-

based theorem provers unsuitable for the task of induction proofs. Proof planning is intended to avoid this object-level, non-strategic search. The current *CLAM* library uses a meta-level *wave* annotation and a technique known as *rippling* to guide its search for induction proofs. This is often good enough to eliminate search (or rather backtracking) altogether. We give an illustration of this using the theorem of the associativity of `plus`. This asserts that

$$\vdash \forall x : \mathtt{pnat}.\forall y : \mathtt{pnat}.\forall z : \mathtt{pnat}.x + (y + z) = (x + y) + z$$

If we perform primitive induction on **x** we have to prove two subgoals, a non-recursive *base case* and a recursive *step case*:

$$\vdash \forall y : \mathtt{pnat}.\forall z : \mathtt{pnat}.0 + (y + z) = (0 + y) + z$$

and

$$\begin{array}{l} \forall y : \mathtt{pnat}.\forall z : \mathtt{pnat}.x + (y + z) = (x + y) + z \\ \quad \vdash \forall y : \mathtt{pnat}.\forall z : \mathtt{pnat}.\boxed{s(\underline{x})} + (y + z) = (\boxed{s(\underline{x})} + y) + z \end{array}$$

Notice that we have annotated how the *induction conclusion* differs from the *induction hypothesis* in the step case. We use the wave notation $\boxed{front(\underline{hole})}$ to indicate that the expression differs from one we wish to match it against by the *wave-front*, the structure inside the box but not underlined. The desired expression, the *skeleton*, can be recovered by deleting all structure inside a box apart from that which is underlined (the *wave-hole*). This is the form of annotation used by *CLAM* methods. The objective in the step case subgoal is to manipulate the conclusion, for instance by rewriting, until it matches the induction hypothesis. If we rewrite the conclusion using the definition of + provided by equation (A.2) we get, ignoring quantifiers, successively:

$$\begin{array}{rcl} \boxed{s(\underline{x})} + (y + z) & = & (\boxed{s(\underline{x})} + y) + z \\ \boxed{s(\underline{x + (y + z)})} & = & (\boxed{s(\underline{x})} + y) + z \\ \boxed{s(\underline{x + (y + z)})} & = & \boxed{s(\underline{x + y})} + z \\ \boxed{s(\underline{x + (y + z)})} & = & \boxed{s(\underline{(x + y) + z})} \\ x + (y + z) & = & (x + y) + z \end{array}$$

The final cancellation step used the substitutivity lemma (A.3) for the successor function s. We can see that by rewriting we gradually *ripple out* the wave-fronts, and eventually end up with an expression identical to the induction hypothesis (IH). At this point we can *fertilize* the goal, i.e. use the IH and close the step case subproof. It is the necessity of rippling out the wave-front in a conclusion until we reach a point where fertilization is possible that guides the search. The base case can be proved simply by rewriting the goal using equation (A.1) of the definition of $+$ (*symbolic evaluation*). Thus, the `base_case` and `step_case` methods posited above would look, respectively, something like:

```
REPEAT sym_eval
```

```
(REPEAT wave) THEN fertilize
```

where `sym_eval` performs one step of symbolic evaluation and `wave` performs one step of rippling out.

In order to build a plan in this fashion, *CLAM* requires knowledge of the theory or context in which a theorem is set. In this case this consists of the definition of $+$ and the substitutivity of s given by the theorems:

$$\forall y: \ \texttt{pnat}.0 + y \ = \ y \ \texttt{in pnat} \tag{A.1}$$

$$\forall x, y: \ \texttt{pnat}.\boxed{s(\underline{x})} + y \ = \ \boxed{s(\underline{x + y})} \ \texttt{in pnat} \tag{A.2}$$

$$\forall x, y: \ \texttt{pnat}.x = y \ \texttt{in pnat} \ \rightarrow \ \boxed{s(\underline{x})} = \boxed{s(\underline{y})} \ \texttt{in pnat} \tag{A.3}$$

Note that we have again added annotation. This is part of the analysis performed by *CLAM* as the definitions are loaded although $+$ is defined as an OYSTER term, *CLAM* expects in addition a set of theorems describing its evaluation to be present. Rewrite rules in this form are called *wave rules* since they allow us to see immediately how annotations are changed by the rewriting. Since we use these theorems to perform rewriting we display them as oriented rules, so that rewriting goes in the opposite direction to implication arrows when used at positive positions. E.g. the last of the above theorems would be written as

$$\boxed{s(\underline{x})} = \boxed{s(\underline{y})} \ \texttt{in pnat} :\Rightarrow x = y \ \texttt{in pnat} \tag{A.4}$$

Recursion analysis (Stevens, 1989), a rational reconstruction and extension of the method used in the Boyer and Moore theorem prover (Boyer & Moore, 1979), is a technique which, using the loaded wave rules and a data base of inductions schemas, attempts to find an induction appropriate to the goal. This is implemented as part of the `induction` method mentioned above. Recursion analysis involves examining the functions occurring in a goal, in particular at their recursive argument positions. Each variable which is universally quantified outermost (including appearing as a declaration in the hypothesis) is tested to see if rippling out can take place using one of the loaded induction schemas, and given a corresponding score. Candidate inductions are then tried on a best-first basis according to these scores, backtracking on failure.

Since we are concerned with planning synthesis proofs, we need an additional annotation, that of *potential waves*. In synthesis proofs we will usually need to perform induction on existentially quantified goals. The problem with this is that we do not know in advance the value the eventual witness will take; it may or may not contain a wave w.r.t. the IH. To allow for this uncertainty we mark all occurrences of the existential variable as potential waves using a broken box (see below). We make use of this annotation when rippling existentially quantified goals. Suppose we have as goal:

$$\vdash \forall l : \texttt{pnat list}. a : \texttt{pnat}. \exists x : \texttt{pnat list}. x = \texttt{append}(l, a :: \texttt{nil}) \texttt{ in pnat list}$$

After performing induction on l, we are left with the step case:

$$\begin{array}{l} \forall a : \texttt{pnat}. \exists x : \texttt{pnat list}. x = \texttt{append}(t, a :: \texttt{nil}) \texttt{ in pnat list} \\ \vdash \forall a : \texttt{pnat}. \exists x : \texttt{pnat list}. \boxed{x} = \texttt{append}(\boxed{h :: \underline{t}}, a :: \texttt{nil}) \texttt{ in pnat list} \end{array}$$

The broken box marks the potential wave front. The RHS can be rippled out using the definition of `append`, but rippling is then seemingly blocked.

$$\ldots, a : \texttt{pnat} \vdash \exists x : \texttt{pnat list}. \boxed{x} = \boxed{h :: \underline{\texttt{append}(t, a :: \texttt{nil})}} \texttt{ in pnat list}$$

However, we can use the wave rule

$$\boxed{u :: \underline{v}} = \boxed{u :: \underline{z}} \texttt{ in pnat list} \Rightarrow v = z \texttt{ in pnat list} \qquad \text{(A.5)}$$

the substitutivity lemma for the 2nd argument of the list constructor, to perform an *existential ripple*. Since the variables of the wave rule are universally quantified and x in the goal existentially quantified, it is sound and heuristically promising to rewrite the goal to

$$\ldots \vdash \exists u : \texttt{pnat}, v : \texttt{pnat list}. \boxed{u :: v} = \boxed{h :: \underline{\texttt{append}(t, a :: \texttt{nil})}} \texttt{ in pnat list}$$

and turn the potential into a real wave front which can be rippled out by the wave rule, instantiating u with h. In this case the result is the immediately fertilizable goal:

$$\ldots \vdash \exists v : \texttt{pnat list}. \boxed{v} = \texttt{append}(t, a :: \texttt{nil}) \texttt{ in pnat list}$$

In our synthesis proofs we shall also need two further adornments. We term a position in the goal which is occupied by a universally quantified variable in the hypothesis a *sink* and denote it by $P(\lfloor t \rfloor)$. The idea is that if all terms in the sinks are the same then we can instantiate the hypothesis to match the conclusion at theses positions. Thus, a sink can swallow terms and gives rise to the important heuristic of *rippling in* towards one.

All the definitions above suppose the existence of only one induction hypothesis. But there may be nested inductions or inductions where the step case gives rise to several hypotheses. We therefore need to label both wave holes and sinks according to the hypothesis which generate them. So called *multi-hole* wave fronts and rules will be seen used in chapter 4.

A *CLAM* planner takes as input a goal and attempts to build a *terminating* plan, i.e. one which should result in a proof with no open leaves. A method (see below) is applicable if its input slot matches the goal and if its pre-conditions hold. If this is the case its effects are calculated and the output subgoals returned. The planner can then attempt to solve these recursively. Note that the input and output slots do not have to be strictly object level terms. By including in them meta-level annotation such as waves, methods are able to communicate at the meta-level. By using proof plans to limit the overall strategy which a planner can attempt to build (like `ind_strat` above), the search space can be strictly circumscribed to areas which are likely to be successful.

The above is a greatly simplified description of the current *CLAM* system: it has a large repertoire of methods and submethods for performing such things as case-splits, propositional reasoning and generalisation, and the way in which rippling is performed is now considerably more sophisticated (Bundy *et al*, 1990b; Bundy *et al*, 1993; Ireland, 1992; Yoshida *et al*, 1994). However, *CLAM* is limited only by the library of methods supplied: one is free to use one's own meta-language for talking about proofs, one more appropriate for the particular domain of enquiry.

The notion of waves described here is a very general idea, a formalised notion of term-level analogy. For this reason we should expect to be able to carry out a large part of our synthesis work using the existing *CLAM* method library and extensions thereof. Sometimes, however, it is more reasonable, as in the case of reasoning about decidability or failure, to write methods specialised to deal with these. The resulting method library for tactic synthesis is described in detail in chapter 4.

A.2.2 Methods

It was mentioned above that a *CLAM* method consists of a tactic plus its specification in a metalanguage of the user's choice. To be more concrete, methods are stored in the following form:

```
method(Name(Args),
       Input,
       Pre\_conditions,
       Effects,
       Output,
       Tactic,
      ).
```

It has also been described how the *CLAM* planners make use of methods as *pseudo-tactics*: i.e. use them to accurately simulate at the meta-level the behaviour of object-level tactics. Besides this procedural interpretation, we can also give methods a declarative reading: there is a method called `Name` with arguments `Args`;

the goal is unified with Input,

to Effects should also be successful and ... ditions is successful then a call Output; running Tactic on a real goal corresponding to ...goals are unified with and result in the subgoals described by Output. ... successful

An example is provided by one of the clauses comprising a simplified wave method:

```
method(wave(Pos,[Rule,Dir]),
       [...],
       [matrix(Vars,Matrix,G),
        wave_rule(Rule,Dir,L:=>R),
        exp_at(Matrix,Pos,L)
       ],
       [replace(Pos,R,Matrix,NewMatrix),
        matrix(Vars,NewMatrix,NewG)
       ],
       [H==>NewG],
       wave(Pos,[Rule,Dir])
      ).
```

This method, which is iterated to exhaustion in step_case, performs a single ripple using any applicable wave rule in *CLAM*'s data base. It can be paraphrased as: this method applies if we can remove all the universal quantifies from the goal (G) to leave a matrix (Matrix), find a wave-rule (L:=>R), and a position (Pos) which unifies with the latter's LHS. If the LHS is replaced with the now-instantiated RHS, and the resulting matrix (NewMatrix) is requantified, there is a single remaining subgoal (H==>NewG). The identically named tactic should perform the corresponding object level subproof.

The aim of this work is, given the specification part of a method (the four middle slots), to synthesize a tactic satisfying the specification in the above sense. For the reasons given in the first half of this chapter we have chosen to do this, not in the native Prolog of *CLAM*, but using the main target logic of *CLAM*, OYSTER. The representation we use to carry this out is described fully in the chapter 3.

We represent the functions and predicates of the *CLAM* method language using type theoretic analogues and describe them with appropriate theorems and wave rules as was done above for `plus`. In this way we develop a *theory of tactics*. We do not, however, attempt a full reconstruction of tactics. This would be extremely complex, involving notions of proof trees, applicability of inference rules, binding operators, etc. as in (Knoblock & Constable, 1986), and we would have little hope of automating a significant portion of the synthesis proofs within it. We choose another route, detailed in the next chapter, of concentrating solely on rewrite tactics. We feel this has the advantages of:

Simplicity There is exactly one goal and one "subgoal" for all tactics, viz. the term before and after rewriting.

Usefulness Many of the interesting methods of *CLAM* consist solely of rewriting, e.g. rippling.

Portability If we had chosen to model the OYSTER logic closely this would have made our synthesis methods and synthetic tactics very specific, whereas most logics have a notion of equality/equivalence and substitution.

Implementation There is an existing system (Howe, 1988a), a simplified version of which we present in the next chapter, which allows us to use the synthetic tactics directly in the OYSTER logic.

Proof theory Our proofs of correctness of the synthetic tactic eventually "ground out" as equality goals in the OYSTER object logic. Thus we can use the existing *CLAM* tools for significant parts of proof.

Of course, there are the drawbacks that we cannot synthesize directly tactics which return multiple subgoals and that we only attempt to rewrite first-order terms of type `pnat`. We suggest in section ?? how we could in principle redo our programme using a mechanism more like Howe's.

Recalling the Curry-Howard isomorphism described previously, we need, in order to synthesize a tactic appropriate to a method, to prove a goal of the form:

$$\vdash \forall i : input.\texttt{pre\_conditions}(i) \rightarrow \exists o : output.effects(i, o) \wedge i \sim o$$

Here, as well as our representation of the method's specification we include a correctness goal $i \sim o$. This is necessary to justify the use of the synthesis proof by the reflection mechanism of chapter 3 and essentially says that i and o have the same meaning. The extract from a proof of this kind will have the form $\lambda i.\lambda p. < o, \_ >$ and is obviously not what one would normally think of as a tactic. We propose two ways to use such an extract: via the reflection mechanism described in chapter 3; and via an informal translation of the extract to a Prolog program (chapter 7: optional).

A.2.3 Summary

We have chosen to synthesize tactics in a deductive system. We choose to use the flexible approach to program construction provided by the OYSTER type theory, and to directly synthesize tactics via OYSTER's built-in Curry-Howard extraction mechanism. We can use the synthetic tactic either directly, via a reflection mechanism, or indirectly via translation to Prolog. Synthesis is identified with theorem proving in the class of $\forall\exists$ goals, where for simplicity's sake we consider only rewrite tactics. We automate synthesis using the *CLAM* proof planner, providing extensions to the method repertoire to deal specifically with this problem domain. By these means we are able to tell a high-level, strategic story, i.e. provide a proof plan, for tactic synthesis.

A.3 Clam glossary

The definitive reference is (vanHarmelen *et al*, 1993).

annotated term A term of the object level (q.v.) logic annotated with meta level (q.v.) information, e.g. wave fronts, wave holes and sinks (q.v.).

base case See induction.

cancellation rule These are wave rules (q.v.) derived from substitutivity lemmas which have the effect of removing wave fronts altogether. E.g. the cancellation rule derived from the substitutivity lemma

$$\forall x, y: \ \texttt{pnat}.x = y \ \texttt{in pnat} \rightarrow \mathcal{C}(x) = \mathcal{C}(y) \ \texttt{in} \ T$$

is

$$\forall x, y: \ \texttt{pnat}.\boxed{\mathcal{C}(\underline{x})} = \boxed{\mathcal{C}(\underline{y})} \ \texttt{in} \ T :\Rightarrow x = y \ \texttt{in pnat}$$

coloured rippling When complicated induction schemas (q.v.) are used there may be several induction hypotheses in a step case (q.v.). To distinguish which parts of the induction conclusion (q.v.) correspond to which induction hypothesis, the various wave-fronts are given indexes (called colours). The rippling process (q.v.) is thus given extra information towards guiding wavefronts to positions allowing fertilization (q.v.). See also (Yoshida, 1993; Yoshida *et al*, 1994).

computational induction Induction (q.v.) based on the recursion scheme of a total function. In the case of simple functions this may be constructor induction, but in general it is destructor induction. C.f. structural induction.

conditional wave rule A wave rule of the form

$$\mathcal{C} \rightarrow \mathcal{L} :\Rightarrow \mathcal{R}$$

where $\mathcal{C}$ is a condition consisting of a conjunction of atoms or negated atoms.

constructor induction Induction (q.v.) based on a schema (q.v.) which results in step cases where the induction conclusion differs from the hypothesis(es) by the *addition* of constructor symbols. The induction hypotheses are instances of the original conclusion. C.f. destructor induction.

creational wave rule A wave rule (q.v.) which has the effect of introducing wave fronts where there were none before. Typically this is done using a constructor-destructor pair. If c and d are such a pair then from the equation

$$\forall x : T.d(c(x)) = x \text{ in } T$$

we get the creational wave rule

$$\forall x : T.x :\Rightarrow \boxed{d(c(\underline{x}))}$$

Creational wave rules are used in particular in cross-fertilization (q.v.).

cross wave rule See creational wave rule.

definition A library logical object (`def`) containing the definition of an object level term. To define a function called `fun`, this takes the form

```
fun(x_1,...,x_n) <==> term
```

where `x_1,...,x_n` are the formal arguments and `term` is an object level term whose free variables are amongst `x_1,...,x_n`. When a definition for `fun` is loaded (q.v.), equations (q.v.) with names of the form `funN` are loaded automatically and are expected to provide an unfolded definition of `fun` based on its recursion scheme.

destructor induction Induction (q.v.) based on a schema (q.v.) which results in step cases where the induction conclusion is an instance of the original conclusion, and the induction *hypothesis(es)* differs from this by the addition of destructor symbols. This is a more general than constructor induction (q.v.) since arbitrary well-founded measures may be used, with functions decreasing in them as destructors. Constructor induction may be transformed into destructor induction by rippling across (q.v.).

effects One of the slots of a method (q.v.); synonymous with post-conditions (q.v.).

equation A library logical object (`eqn`) used to store proofs of equations describing the recursion scheme of a defined function. See also definition.

erasure The *erasure* of an annotated term is that term with all annotation removed, e.g. the erasure of $\boxed{s(\underline{x})} + y$ is $s(x) + y$. An erasure is always an object level term.

fertilization Fertilization is the point in the step case of an inductive proof at which the induction hypothesis (q.v.) is used. Fertilization can be classified as either strong or weak (q.v.).

hard wave front A wave front (q.v.) which is not soft (q.v.).

hints A mechanism whereby assistance in the form of declarative search restrictions may be given to *CLAM* before or during planning. Typically this is done when *CLAM* fails to find a plan, or never terminates, using one of the standard planners (q.v.). For a detailed reference see (Negrete, 1992; vanHarmelen *et al*, 1993).

induction We encapsulate mathematical induction over inductively defined data types as rules of inference or schemas derived from these rules. E.g. primitive induction over the type of Peano natural numbers (`pnat`) is expressed by the following OYSTER rule:

$$\frac{\mathcal{H} \vdash \mathcal{P}[0/x] \qquad \mathcal{H}, x : \ \mathtt{pnat}, \mathcal{P} \vdash \mathcal{P}[s(x)/x]}{\mathcal{H}, x : \ \mathtt{pnat} \vdash \mathcal{P}} \ \mathrm{elim(on(x))}$$

There are similar rules for types such as integers and lists. The non-recursive premisses are called the *base cases*, and the recursive premisses the *step cases*. In this example the first premiss is a base case and the second a step case. In step cases the instances of the original conclusion appearing in the hypothesis list are called *induction hypotheses*, that appearing in the conclusion the *induction conclusion*.

induction conclusion See induction.

induction hypothesis See induction.

induction schema See induction.

joining (of wave fronts) The opposite of splitting (q.v.).

lemma A logical library object (verb+lemma+) used to store auxiliary proofs required by top level *theorems* (q.v.).

loading (a definition, wave rule etc.) *CLAM* has a library mechanism which allows the saving and loading of various types of logical object: theorems, lemmas, equations, definitions, schemas, (sub)methods and hints. When a definition is loaded its corresponding equations are parsed in order to extract all possible wave rules and ordinary rewrite rules derivable from them, and these are cached to allow efficient look up during planning. *CLAM* can also be directed to parse theorems and lemmas and cache the derived rules. See also *needs file*.

longitudinal wave rule A wave rule (q.v.) in which a wave front (q.v.) is moved, either inward or outward, along a path in the term. C.f. transverse wave rules. This is usually done with the aim of later removing the wave front and strong fertilizing (q.v.), or moving it sufficiently to allow weak fertilization to take place.

meta level The level at which inference is carried out in *CLAM*. This term may refer to annotation, methods, planning, the running of pre- and post-conditions, etc.

method The meta level (q.v.) specification of a tactic. This consists of six parts or slots: a name, an input (goal), an output (list of subgoals), a list of preconditions, a list of post-conditions, and a tactic name. It is intended that the named tactic satisfy the specification.

methodical A connective for combining (sub)method(s) (q.v.) to form new a (sub)method; the method analogue of tacticals. Examples are composition (`then/2`), alternation (`or/2`) and iteration (`iterate/5`).

middle-out reasoning A technique where meta level (q.v.) variables appear in plans, allowing instantiation as a later stage of planning when more information about the form of the proof is available. These variables may include induction schemas and existential witnesses. See also (Bundy *et al.* 1989a).

monochromatic rippling Traditional, single-colour rippling.

monotonicity *CLAM* maintains a hand-coded record of whether certain functions' arguments are monotonic or anti-monotonic w.r.t. to given orderings, or neither.

multi-hole rippling See coloured rippling.

"needs" file In order for planning to go through completely automatically, *CLAM* sometimes requires the loading of rules in addition to those loaded (by convention) with definitions. These extra rules are detailed in a file called `needs.pl` on a logical object-by-logical object basis.

object level The formal level at which proofs are carried out. At present this is the theorem proving environment OYSTER, a sequent-style presentation of intuitionist type theory (see (Horn, 1988) or section A.1).

plan execution Each method (q.v.) when called instantiates its tactic slot with the name of a suitable tactic (q.v.).

plan A tactic produced by a planner (q.v.). These tactics act as very-high-level descriptions of a proof.

planner A meta level program which attempts to find a plan capable of solving the goal. There are different planners corresponding to the different search techniques employed in the planning (q.v.). Currently there are depth-first, breadth-first, iterative-deepening and best-first planners available in *CLAM*.

planning The search process where a method (q.v.) is selected from those available and its applicability to the current goal tested by running its pre-conditions (q.v.). If these succeed, the post-conditions (q.v.) are run to generate the subgoal(s). If there are no subgoals planning has succeeded; otherwise planning is carried out recursively for all subgoals.

polarity The conventional proof-theoretic definition of polarity (see e.g. (Girard, 1987)). If rewriting at a positive position in a goal, implications are used right-to-left. At negative positions the direction is reversed. See also *wave rule*.

post-conditions A declarative description of the output slot of an applicable method. In practice, the post-conditions are used entirely procedurally to build the subgoal(s), hence the synonym *effects*.

potential wave front (synonym of soft wave front) A meta level annotation (q.v.) indicating where a wave front (q.v.) *may* be introduced due to the presence of an existentially quantified variable the existential variable may be partially instantiated to a wave front surrounding a new, existentially quantified wave hole. This is important since it can motivate otherwise invisible inductions. E.g. in the goal

$$\ldots \vdash \exists y : \ \texttt{pnat}.\boxed{\boxed{y}} = \boxed{s(\underline{x})} \ \texttt{in pnat}$$

y may be instantiated to $s(y')$ giving

$$\ldots \vdash \exists y' : \ \texttt{pnat}.\boxed{s(\boxed{y'})} = \boxed{s(\underline{x})} \ \texttt{in pnat}$$

Traditionally potential waves are annotated as $\boxed{\text{p-wave}}$.

pre-conditions The meta level (q.v.) conditions, usually w.r.t. the input (q.v.), which must hold for a given method to be applicable.

primitive induction Induction (q.v.) based on a primitive inference rule of the object level (q.v.).

proof plan A high-level, specialised strategy (method (q.v.)) for solving a particular type of problem, usually built from smaller (sub)methods. An example is the `ind_strat/1` method specialised for inductive proofs.

recursion analysis (now known as ripple analysis) Definitions and additional lemmas, and the resulting wave rules, describe the forms of recursion employed in the functions present in a goal. Candidate variables on which to perform induction can be deduced from the effective quantification prefix of the goal. By attempting to ripple out (q.v.) putative wave fronts indicated by the wave rules applying to functions in the goal, and scoring each according to a number of heuristic criteria, inductions schemas which have a higher likelihood of success can be chosen. This is *recursion analysis*. See also (Bundy *et al*, 1989b).

reduction rule A rule which removes a constant expression or is a wave-rule where the wave-front is a type constructor, and used in during symbolic evaluation (q.v.).

ripple analysis Synonym of recursion analysis.

rippling The process of rewriting an annotated goal using wave rules (q.v.). In order that this process terminate wave fronts are restricted in their movements: outward movement (via longitudinal wave rules (q.v.)) is allowed initially, possibly followed by one change of direction (via a transverse wave rule (q.v.)) and inward movement (again using longitudinal rules). The first phase is known as *rippling out* and may allow fertilization to take place. The second phase is called *rippling in* and takes place if fertilization was impossible or only partially successful at the end of the first phase. Rippling in is restricted to cases where a sink (q.v.) lies below the wave front.

rippling in See rippling.

rippling out See rippling.

sink Sinks correspond to universally quantified variables in the induction hypothesis (q.v.). Since such variables can be instantiated to any suitably typed term sinks are capable of "swallowing" wave fronts (q.v.). Since a universal variable may occur more than once in an IH care must be taken that sinks corresponding to the same variable are instantiated with the same term. Traditionally sinks are annotated as $\lfloor \text{sink} \rfloor$.

skeleton The term which is the result of removing all the wave fronts from an annotated term (q.v.).

soft wave front Synonym of *potential wave front.*

splitting (of wave fronts) A wave front consisting of two or more constructors can be split up into several smaller, composed wave fronts. E.g. the compound wave front $\boxed{c(d(\underline{x}))}$ can be *split* to give $\boxed{c(\boxed{d(\underline{x})})}$. While semantically identical, the distinct annotations must be taken into account during syntactic operations such as rippling. C.f. joining.

step case See induction.

strong fertilization The case of fertilization (q.v.) in which the whole or part of the induction conclusion matches the whole of the induction hypothesis, allowing it to be rewritten simply to `true`. C.f. weak fertilization.

structural induction Induction based on the structure of the data type. For freely generated data types this will be constructor induction (e.g. Peano natural numbers, lists). Otherwise, destructor-style induction will usually be required.

submethod A *submethod* is a method (q.v.) which cannot be used at the top level of planning but which may be called by other methods and submethods.

symbolic evaluation The use of traditional (non-annotated) rewrite rules. Typically these are theorems describing the non-recursive cases of a definition.

tactic A program written in a meta level (q.v.) language which carries out object level (q.v.) proofs. See section 2.2 for a more detailed description.

tactical A higher-order function for combining tactics into new tactics. See section 2.2 for a more detailed description.

theorem A library logical object (`thm`) used to store top level proofs. C.f. lemmas and equations.

transverse wave rule A wave rule (q.v.) in which a wave front (q.v.) is moved between positions which do not share a common path in the term. C.f. longitudinal wave rules. These are usually applied in order to loose the wave front in a sink (q.v.) by later rippling in (q.v.).

wave direction A wave front can be in the process of being rippled out (q.v.) or rippled in (q.v.). Since this information is needed for the termination of rippling it is added as an extra piece of annotation. Outward-moving fronts are indicated $\boxed{\text{front}(\underline{\text{hole}})}^{\uparrow}$, inward-moving as $\boxed{\text{front}(\underline{\text{hole}})}^{\downarrow}$.

wave front Part of an annotated term (q.v.) which it is desirable to move or remove. This usually corresponds to the difference or "extra syntax" between an induction conclusion and hypothesis (q.v.). A wave front is part of the erasure (q.v.) but not the skeleton (q.v.). Traditionally wave fronts and holes are annotated as $\boxed{\text{front}(\underline{\text{hole}})}$. C.f. wave hole.

wave hole Part of an annotated term (q.v.) which it is desirable to keep. This is usually part of an induction conclusion (q.v.) which has a corresponding part in the induction hypothesis (q.v.). A wave hole is part of both the erasure and the skeleton (q.v.). C.f. wave front.

wave parsing The process carried out as definitions, theorems and lemmas are loaded into $CLAM$ and checked for having the form of a wave rule. If so, an efficient representation of the rule is cached.

wave rule A formula expressing a relation (typically equality or logic implication) between two terms which has been parsed and annotated to show the movement of wave fronts, wave holes and sinks (q.v.). If the relation between terms is logical implication ($\rightarrow$) the rule is called *implicative*. Wave rules are usually written backwards to show the effect of using them as rewrite rules at positions of positive polarity (q.v.). E.g. equation (A.3) would be written as wave rule (A.4). See also conditional wave rule.

weak fertilization The case of fertilization (q.v.) where the induction hypothesis is an equation or implication only one side of which matches a subterm of the induction conclusion. The induction hypothesis is then used as a rewrite in the traditional way and the resulting subgoal attempted afresh. C.f. strong fertilization.

Appendix B

Predicate calculus in OYSTER

$$\frac{H >> C \qquad H, x : C >> G}{H >> G}\ \text{seq}(C, \text{new[x]}) \qquad \frac{H, H' >> G}{H, x : C, H' >> G}\ \text{thin(x)}$$

$$\frac{}{H, x : A, H' >> A}\ \text{hyp(x)} \qquad \frac{}{H, x : \texttt{void}, H' >> G}\ \bot\text{-elim(x)}$$

$$\frac{H >> A \qquad H >> B}{H >> A \# B}\ \wedge\text{-intro}$$

$$\frac{H, x : A \# B, H', u : A, v : B >> G}{H, x : A \# B, H' >> G}\ \wedge\text{-elim(x,new[u,v])}$$

$$\frac{H, x : A >> B}{H >> A \rightarrow B}\ \rightarrow\text{-intro(new[x])}$$

$$\frac{H, x : A \rightarrow B, H' >> A \qquad H, x : A \rightarrow B, H', y : B >> G}{H, x : A \rightarrow B, H' >> G}\ \rightarrow\text{-elim(x,new[y])}$$

$$\frac{H >> A}{H >> A|B}\ \vee\text{-intro(left)} \qquad \frac{H >> B}{H >> A|B}\ \vee\text{-intro(right)}$$

$$\frac{H, x : A|B, H', u : A >> G \qquad H, x : A|B, H', v : B >> G}{H, x : A|B, H' >> G}\ \vee\text{-elim(x,new[u,v])}$$

$$\frac{H >> P[y/x]}{H >> \forall x : T.P}\ \forall\text{-intro(new[y])}$$

$$\frac{H, y : \forall x : T.P, H', z : P[a/x] >> G}{H, y : \forall x : T.P, H' >> G}\ \forall\text{-elim(y,on(a),new[z])}$$

$$\frac{H >> P[a/x]}{H >> \exists x : T.P}\ \exists\text{-intro(a)}$$

$$\frac{H, y : \exists x : T.P, H', v : P[u/x] >> G}{H, y : \exists x : T.P, H' >> G}\ \exists\text{-elim(y,new[u,v])}$$

Appendix C

Lemmas

C.1 Lemma 1

This proves the lemma showing the limited number of intuitionistically distinct formulas containing only one propositional variable and that only positively (or negatively). I.e. that this subclass of the Heyting algebra (which is not a sub-algebra) is finite.

Suppose $\mathcal{F}$ is a propositional formula containing only the propositional variable B and that this occurs only positively (resp. negatively). Then $\mathcal{F}$ is intuitionistically equivalent to one of

$$\top \qquad \bot \qquad B \qquad \neg\neg B$$

(resp.

$$\top \qquad \bot \qquad \neg B$$

)[1].

[1] Throughout $\bot$, $\top$, and $\neg\mathcal{F}$ abbreviate resp. `void`, `void` $\rightarrow$ `void`, and $\mathcal{F} \rightarrow$ `void`.

Proof is by induction over formulas. Because of the connective $\rightarrow$, where B occurs positively (resp. negatively) iff it occurs positively (resp. negatively) in the succeedent and negatively (resp. positively) in the antecedent, our induction needs to contain simultaneously the positive and negative cases.

Base case: $\mathcal{F}$ is atomic

The only possibilities are that $\mathcal{F}$ is one of the following: $\top$, $\bot$, or B.

Step case: $\mathcal{F}$ is compound

In the cases of the connectives $\wedge$ and $\vee$ the equivalences provided by the following tables suffice:

$\wedge$	$\top$	$\bot$	B	$\neg\neg B$
$\top$	$\top$	$\bot$	B	$\neg\neg B$
$\bot$	$\bot$	$\bot$	$\bot$	$\bot$
B	B	$\bot$	B	B
$\neg\neg B$	$\neg\neg B$	$\bot$	B	$\neg\neg B$

$\vee$	$\top$	$\bot$	B	$\neg\neg B$
$\top$	$\top$	$\top$	$\top$	$\top$
$\bot$	$\top$	$\bot$	B	$\neg\neg B$
B	$\top$	B	B	$\neg\neg B$
$\neg\neg B$	$\top$	$\neg\neg B$	$\neg\neg B$	$\neg\neg B$

$\wedge$	$\top$	$\bot$	$\neg B$
$\top$	$\top$	$\bot$	$\neg B$
$\bot$	$\bot$	$\bot$	$\bot$
$\neg B$	$\neg B$	$\bot$	$\neg B$

$\vee$	$\top$	$\bot$	$\neg B$
$\top$	$\top$	$\top$	$\top$
$\bot$	$\top$	$\bot$	$\neg B$
$\neg B$	$\top$	$\neg B$	$\neg B$

In the case of the connective $\rightarrow$ we have that B occurs positively (resp. negatively) in $\mathcal{E} \rightarrow \mathcal{F}$ iff B occurs positively (resp. negatively) in $\mathcal{E}$ and negatively (resp. positively) in $\mathcal{F}$. These possibilities are covered by the following two tables:

$\rightarrow$	$\top$	$\bot$	B	$\neg\neg B$
$\top$	$\top$	$\bot$	B	$\neg\neg B$
$\bot$	$\top$	$\top$	$\top$	$\top$
$\neg B$	$\top$	$\neg\neg B$	$\neg\neg B$	$\neg\neg B$

$\rightarrow$	$\top$	$\bot$	$\neg B$
$\top$	$\top$	$\bot$	$\neg B$
$\bot$	$\top$	$\top$	$\top$
B	$\top$	$\neg B$	$\neg B$
$\neg\neg B$	$\top$	$\neg B$	$\neg B$

Q.E.D.

C.2 Lemma 2

This proves the simple lemma that our deduction system of appendix B can be reduced to one where the hyp rule applies only to atomic formulas.

Proof is by induction on the formula used in the hyp rule, showing that we can replace an instance of a non-atomic hyp with a subproof all of whose hyp's are strictly simpler.

Suppose we have an instance of the hyp rule of the form

$$\frac{}{\Gamma, x : \mathcal{A}, \Gamma' >> \mathcal{A}}\ \text{hyp(x)}$$

Performing induction on $\mathcal{A}$ we have the following cases.

$\mathcal{A} \equiv \mathcal{B}\#\mathcal{C}$

We replace the instance with

$$\dfrac{\dfrac{}{\ldots, u : \mathcal{A}, v : \mathcal{B} >> \mathcal{A}}\,\text{hyp(u)} \quad \dfrac{}{\ldots, u : \mathcal{A}, v : \mathcal{B} >> \mathcal{B}}\,\text{hyp(v)}}{\dfrac{\ldots, u : \mathcal{A}, v : \mathcal{B} >> \mathcal{A}\#\mathcal{B}}{\Gamma, x : (\mathcal{A}\#\mathcal{B}), \Gamma' >> \mathcal{A}\#\mathcal{B}}\,\wedge\text{-elim(x,new[u,v])}}\,\wedge\text{-intro}$$

$\mathcal{A} \equiv \mathcal{B} \to \mathcal{C}$

We replace the instance with

$$\dfrac{\dfrac{}{\ldots, y : \mathcal{A} >> \mathcal{A}}\,\text{hyp(y)} \quad \dfrac{}{\ldots, y : \mathcal{A}, z : \mathcal{B} >> \mathcal{B}}\,\text{hyp(z)}}{\dfrac{\Gamma, x : (\mathcal{A} \to \mathcal{B}), \Gamma', y : \mathcal{A} >> \mathcal{B}}{\Gamma, x : (\mathcal{A} \to \mathcal{B}), \Gamma' >> \mathcal{A} \to \mathcal{B}}\,\to\text{-intro(new[y])}}\,\to\text{-elim(x,new[z])}$$

$\mathcal{A} \equiv \mathcal{B}|\mathcal{C}$

We replace the instance with

$$\dfrac{\dfrac{\dfrac{}{\ldots, u : \mathcal{A} >> \mathcal{A}}\,\text{hyp(u)}}{\ldots, u : \mathcal{A} >> \mathcal{A}|\mathcal{B}}\,\vee\text{-intro(left)} \quad \dfrac{\dfrac{}{\ldots, v : \mathcal{B} >> \mathcal{B}}\,\text{hyp(v)}}{\ldots, v : \mathcal{B} >> \mathcal{A}|\mathcal{B}}\,\vee\text{-intro(right)}}{\Gamma, x : (\mathcal{A}|\mathcal{B}), \Gamma' >> \mathcal{A}|\mathcal{B}}\,\vee\text{-elim(x,new[u,v])}$$

$\mathcal{A} \equiv \forall x : T.\mathcal{B}$

Here the instance is

$$\dfrac{}{\Gamma, y : \forall x : T.\mathcal{B}, \Gamma' >> \forall z : T.\mathcal{B}'}\,\text{hyp(y)}$$

where $\mathcal{B}'[x/z] \equiv_\alpha \mathcal{B}$. This is replaced by

$$\dfrac{\dfrac{\dfrac{}{\ldots, v : \mathcal{B}[u/x] >> \mathcal{B}'[u/z]}\,\text{hyp(v)}}{\Gamma, y : \forall x : T.\mathcal{B}, \Gamma' >> \mathcal{B}'[u/z]}\,\forall\text{-elim(y,on(u),new[v])}}{\Gamma, y : \forall x : T.\mathcal{B}, \Gamma' >> \forall z : T.\mathcal{B}'}\,\forall\text{-intro(new[u])}$$

Note that the new hyp use is justifies since $\mathcal{B}[u/x] \equiv_\alpha \mathcal{B}'[u/z]$.

$\mathcal{A} \equiv \exists x : T.\mathcal{B}$

Here the instance is

$$\frac{}{\Gamma, y : \exists x : T.\mathcal{B}, \Gamma' >> \exists z : T.\mathcal{B}'}\ \text{hyp(y)}$$

where $\mathcal{B}'[x/z] \equiv_\alpha \mathcal{B}$. This is replaced by

$$\dfrac{\dfrac{\dfrac{}{\ldots, v : \mathcal{B}[u/x] >> \mathcal{B}'[u/z]}\ \text{hyp(v)}}{\ldots, v : \mathcal{B}[u/x] >> \exists z : T.\mathcal{B}'}\ \exists\text{-intro(u)}}{\Gamma, y : \exists x : T.\mathcal{B}, \Gamma' >> \exists z : T.\mathcal{B}'}\ \exists\text{-elim(y,new[u,v])}$$

Note that the new hyp use is justifies since $\mathcal{B}[u/x] \equiv_\alpha \mathcal{B}'[u/z]$.

C.2.1 Notes

We could refine the proof further to ensure that only propositional *variables* are used in hyp rules by replacing instances of hyp(`void`) with $\perp$-elim.

In dealing with the two quantifier cases we have noted that the hyp rule only requires the fertilizer and fertilizee to be identical upto α-convertibility, and we showed that the manipulation of bound variables occurring in these cases preserves that property. This should have been done for the other cases considered, but since no bound variables are affected it was not mentioned.

The process described above is compatible with cut-elimination and our revised proof system S' since we can obtain a cut-free proof in S' with only atomic hyp's by carrying out (in S') first cut-elimination and then the process described above (i.e. no cuts are introduced).

C.3 Lemma 3: WRONG

We define a proof system S' similar to our original system S and show that they are equivalent.

Our proof system S is defined in appendix B (\ref{pred-calc-app}) of the thesis. To obtain system S' replace the $\wedge$-elim, $\rightarrow$-elim, $\vee$-elim, and $\exists$-elim with resp.

$$\frac{\Gamma, \Gamma', u : \mathcal{A}, v : \mathcal{B} >> \mathcal{G}}{\Gamma, x : (\mathcal{A} \# \mathcal{B}), \Gamma' >> \mathcal{G}} \wedge\text{-elim}'(\text{x,new[u,v]})$$

$$\frac{\Gamma, \Gamma' >> \mathcal{A} \qquad \Gamma, \Gamma', y : \mathcal{B} >> \mathcal{G}}{\Gamma, x : (\mathcal{A} \rightarrow \mathcal{B}), \Gamma' >> \mathcal{G}} \rightarrow\text{-elim}'(\text{x,new[y]})$$

$$\frac{\Gamma, \Gamma', u : \mathcal{A} >> \mathcal{G} \qquad \Gamma, \Gamma', v : \mathcal{B} >> \mathcal{G}}{\Gamma, x : \mathcal{A} | \mathcal{B}, \Gamma' >> \mathcal{G}} \vee\text{-elim}'(\text{x,new[u,v]})$$

and

$$\frac{\Gamma, \Gamma', v : \mathcal{P}[u/x] >> \mathcal{G}}{\Gamma, y : \exists x : T.\mathcal{P}, \Gamma' >> \mathcal{G}} \exists\text{-elim}'(\text{y,new[u,v]})$$

We show that S and S' are equivalent, i.e. that a sequent can be proved in one iff it can be proved in the other, by giving derived rules for each system in the other. Since the two systems differ only in the elimination rules described above these are the only ones we will show.

First, assume we are working in the system S. Then the above rules can be derived simply by thinning. All derivations have the same basic shape as that for $\rightarrow$-elim$'$:

$$\left.\frac{\dfrac{\Gamma, \Gamma' >> \mathcal{A}}{\Gamma, x : (\mathcal{A} \rightarrow \mathcal{B}), \Gamma' >> \mathcal{A}}\,\text{thin(x)} \qquad \dfrac{\Gamma, \Gamma', y : \mathcal{B} >> \mathcal{G}}{\Gamma, x : (\mathcal{A} \rightarrow \mathcal{B}), \Gamma', y : \mathcal{B} >> \mathcal{G}}\,\text{thin(x)}}{\Gamma, x : (\mathcal{A} \rightarrow \mathcal{B}), \Gamma' >> \mathcal{G}} \rightarrow\text{-elim(x,new[y])}\right\} \rightarrow\text{-eli}$$

Equally, we can do the converse using cuts. All derivations have the same basic shape as that for $\rightarrow$ -elim:

$$\dfrac{\dfrac{}{\Gamma, \Gamma', w : (\mathcal{A} \rightarrow \mathcal{B}) >> \mathcal{A} \rightarrow \mathcal{B}}\,\text{hyp(w)} \quad \dfrac{\Gamma, \Gamma', x : (\mathcal{A} \rightarrow \mathcal{B}) >> \mathcal{A}}{\Gamma, \Gamma', w : (\mathcal{A} \rightarrow \mathcal{B}), x : (\mathcal{A} \rightarrow \mathcal{B}), >> \mathcal{A}}\,\text{thin(w)}}{\dfrac{\Gamma, \Gamma', w : (\mathcal{A} \rightarrow \mathcal{B}) >> \mathcal{A}}{\dfrac{}{\Gamma, x : (\mathcal{A} \rightarrow \mathcal{B}}}\,\text{seq}(\mathcal{A} \rightarrow}$$

$\Gamma, x : (.$

Note that three cuts were introduced here. The derivation is complicated by the need to preserve variable names amongst the declarations. If this requirement can be overlooked then only one cut is required.

We have proved elsewhere cut-elimination theorems for both S and S' which, with the above derivations, allows us to infer the existence of transformations from (arbitrary) proofs in one system to cut-free proofs in the other.

C.3.1 Notes

We have been careful over the use of declaration names in the proofs above since these are important in a constructive logic. If one assumes that derivations in S or S' are part of a larger, general OYSTER proof then one must take care over how the thinning rule and the new elimination rules are represented since other hypotheses may be affected. One should consider our presentations of predicate logic in OYSTER merely as abbreviations of standard proofs with thinning etc. implemented by some form of meta-level elision.

We have concentrated on the $\rightarrow$ -elim$'$ in the above proof, though other elimination rules as indicated are changed in S'. However, it appears that we only ever need the change to the $\rightarrow$ -elim rule in the proofs which apply this lemma.

C.4 Lemma 4: WRONG

Here we prove a technical lemma necessary for a case in the main correctness proof (lemma 9).

If

$$\Gamma, \mathcal{C}[K] \to \mathcal{D}, \mathcal{C}[A] >> \mathcal{D}$$

where, in addition the assumptions made in lemma 9, we also have that $\mathcal{D}$ is atomic and does not occur in Γ, $\mathcal{C}[K]$ or $\mathcal{C}[A]$, then

$$\Gamma, \mathcal{C}[A] >> \mathcal{C}[K]$$

C.4.1 Proof

Proof is by induction on the proof of the initial sequent. By lemma 3 we may assume that this is a cut-free proof in S'. Since $\mathcal{D}$ is atomic there can be no intro rules on any main path to the root (i.e. a path which only passes through the major premisses of any elimination rule applications). By a simple induction it can be shown that every sequent on a main path (before a possible $\to$ -elim$'$ on $\mathcal{C}[K] \to \mathcal{D}$) has the form

$$\Delta, \mathcal{C}[K] \to \mathcal{D}, \mathcal{C}[A] >> \mathcal{D}$$

where $\Gamma >> \Delta$ and $\Delta \not>> \mathcal{D}$.

If there were no $\to$ -elim$'$ on $\mathcal{C}[K] \to \mathcal{D}$ then we would have an infinite branch in the proof: contradiction. Therefore, at some point we have an $\to$ -elim$'$ on $\mathcal{C}[K] \to \mathcal{D}$, i.e.

$$\frac{\begin{array}{cc}\vdots \pi & \vdots \\ \Delta, \mathcal{C}[A] >> \mathcal{C}[K] & \Delta, \mathcal{C}[A], \mathcal{D} >> \mathcal{D}\end{array}}{\Delta, \mathcal{C}[K] \to \mathcal{D}, \mathcal{C}[A] >> \mathcal{D}} \to \text{-elim}'$$

From π we have $\Delta, \mathcal{C}[A] >> \mathcal{C}[K]$ and, since $\Gamma >> \Delta$, $\Gamma, \mathcal{C}[A] >> \mathcal{C}[K]$ as desired.

C.5 Lemma 5: cut-elimination theorem for S': WRONG

The proof systems S and S' for representing intuitionistic, first-order predicate logic in NuPRL are described in appendix B of the thesis (`\ref{pred-calc-app}`) and lemma 3 of this series. We give here a brief proof of the cut-elimination theorem for system S' since it is used indirectly in the proof of CSR correctness.

The proof given here is based on that in (J.-Y. Girard & Taylor, 1989). Since the formulation used in our system S has "persistent" hypotheses, i.e. they are not removed by elimination rules, and differs in other ways from the standard Gentzen presentation we have formulated the cut-reduction lemma as follows.

First we must define a notion of one hypothesis list being included within another. Since the definition of a valid hypothesis list in NuPRL is complicated enough as it is, we define inclusion as: for two valid hypothesis lists Γ and Δ, we say Γ is included in Δ ($\Gamma \subseteq \Delta$) iff there is a sequence of applications of the thin rule leading from $\Gamma >>$ `void` to $\Delta >>$ `void`. Such a sequence of thins will be displayed as:

$$\frac{\Gamma >> \texttt{void}}{\Delta >> \texttt{void}}$$

Note that we regard the various permutations of a hypothesis list as identical as long as they are valid (some perms maybe be invalid because of bound variable scoping). We do, however, require that bound variable names be preserved since these may be important in a constructive proof.

Now we can state the cut-reduction lemma: let $\Gamma >> \mathcal{C}$ and $\Gamma', x : \mathcal{C} >> \mathcal{G}$ be sequents with proofs π and π' and such that their degrees $d(\pi), d(\pi') < d$, and where $\Gamma, \Gamma' \subseteq \Delta$. Then there is a proof ϖ of the sequent $\Delta >> \mathcal{G}$ with degree $< d$.

The proof is by induction on the sum of the heights of the two proofs, $h(\pi)+h(\pi')$. Suppose π and π' have the forms:

$$\pi\left\{\begin{array}{c}\begin{array}{cc}\vdots\,\pi_1 & \vdots\,\pi_m\\ \Gamma_1 >> \mathcal{Q}_1 \;\ldots\; & \Gamma_m >> \mathcal{Q}_m\end{array}\\ \hline \Gamma >> \mathcal{C}\end{array}\right.$$

and

$$\pi'\left\{\begin{array}{c}\begin{array}{cc}\vdots\,\pi'_1 & \vdots\,\pi'_n\\ \Gamma'_1 >> \mathcal{R}_1 \;\ldots\; & \Gamma'_n >> \mathcal{R}_n\end{array}\\ \hline \Gamma', x:\mathcal{C} >> \mathcal{G}\end{array}\right.$$

There are several subcases.

π is an axiom

I.e. we have

$$\pi\left\{\begin{array}{c}\\ \hline \Gamma >> \mathcal{C}\end{array}\right.\text{hyp(y)}$$

where $y:\mathcal{C}\in\Gamma$. Since, therefore, $y:\mathcal{C}\in\Delta$ we can replace every reference in π' to x with one to y giving us $\overline{\pi'}$. For $\overline{\omega}$ we then have

$$\dfrac{\Gamma, y:\mathcal{C} >> \mathcal{G}}{\Delta >> \mathcal{G}}$$

π' is an axiom

I.e. we have

$$\pi'\left\{\begin{array}{c}\\ \hline \Gamma', x:\mathcal{C} >> \mathcal{G}\end{array}\right.\text{hyp(y)}$$

where $y:\mathcal{G}\in\Gamma'$. For $\overline{\omega}$ we then have

$$\dfrac{\dfrac{}{\Gamma' >> \mathcal{G}}\,\text{hyp(y)}}{\Delta >> \mathcal{G}}$$

As (J.-Y. Girard & Taylor, 1989) notes, by giving preference to the first course when both π and π' are axioms we arbitrarily privilege π (unless. of course, $\mathcal{C}\equiv\mathcal{G}$ and $x\equiv y$).

r is a structural rule

I.e. r is a thin:

$$\dfrac{\begin{array}{c}\vdots\,\pi_1\\ \Gamma^* >> \mathcal{C}\end{array}}{\Gamma >> \mathcal{C}}\,\text{thin(y)}$$

We apply the IH to π_1 and π to get a proof $\overline{\omega}$ of $\Delta >> \mathcal{G}$, since $\Gamma^* \subseteq \Gamma \Longrightarrow \Gamma^* \subseteq \Delta$.

r′ is a structural rule

I.e. r′ is a thin:

$$\dfrac{\begin{array}{c}\vdots\,\pi'_1\\ \Gamma' >> \mathcal{G}\end{array}}{\Gamma', x : \mathcal{C} >> \mathcal{G}}\,\text{thin(x)}$$

or

$$\dfrac{\begin{array}{c}\vdots\,\pi'_1\\ \Gamma^*, x : \mathcal{C} >> \mathcal{G}\end{array}}{\Gamma', x : \mathcal{C} >> \mathcal{G}}\,\text{thin(y)}$$

In the first case we have $\overline{\omega}$ directly by:

$$\dfrac{\begin{array}{c}\vdots\,\pi'_1\\ \Gamma' >> \mathcal{G}\end{array}}{\Delta >> \mathcal{G}}$$

In the second case we apply the IH to π and π'_1 to get a proof $\overline{\omega}$ of $\Delta >> \mathcal{G}$, since $\Gamma^* \subseteq \Gamma' \Longrightarrow \Gamma^* \subseteq \Delta$.

r is a logical rule other than a right one of the principal formula $\mathcal{C}$

I.e. r is of the form:

$$\pi\left\{\dfrac{\begin{array}{ccc}\vdots\,\pi_1 & & \vdots\,\pi_m\\ \Gamma_1 >> \mathcal{C} & \ldots & \Gamma_m >> \mathcal{C}\end{array}}{\Gamma >> \mathcal{C}}\,\text{r}\right.$$

Applying the IH to π_i and π' we get:

$$\overline{\omega}\left\{\dfrac{\begin{array}{ccc}\vdots\,\overline{\omega_1} & & \vdots\,\overline{\omega_m}\\ \Delta_1 >> \mathcal{G} & \ldots & \Delta_m >> \mathcal{G}\end{array}}{\Delta >> \mathcal{G}}\,\text{r}\right.$$

r′ is a logical rule other than a left one of the principal formula $x : \mathcal{C}$

I.e. r′ is of the form:

$$\pi' \left\{ \dfrac{\begin{array}{cc} \vdots \pi'_1 & \vdots \pi'_n \\ \Gamma'_1, x : \mathcal{C} >> \mathcal{G} & \ldots \quad \Gamma'_n, x : \mathcal{C} >> \mathcal{G} \end{array}}{\Gamma', x : \mathcal{C} >> \mathcal{G}} \mathrm{r}' \right.$$

Applying the IH to π and π'_i we get:

$$\overline{\omega} \left\{ \dfrac{\begin{array}{cc} \vdots \overline{\omega_1} & \vdots \overline{\omega_n} \\ \Delta_1 >> \mathcal{G} & \ldots \quad \Delta_n >> \mathcal{G} \end{array}}{\Delta >> \mathcal{G}} \mathrm{r}' \right.$$

Note: it wouldn't matter in system S if r′ did affect $x : \mathcal{C}$ since this would remain among the hyp list of the premisses anyway, unlike in system S'.

Both r and r′ are logical rules s.t. r is a right logical rule of $\mathcal{C}$ and r′ is a left logical rule for $x : \mathcal{C}$

We deal here only with the cases that differ from our standard system S; otherwise exactly the same treatment can be given. There are the following subcases depending on the topmost connector of $\mathcal{C}$.

$\mathcal{C} \equiv \mathcal{E} \wedge \mathcal{F}$

$$\pi \left\{ \dfrac{\Gamma >> \mathcal{E} \qquad \Gamma >> \mathcal{F}}{\Gamma >> \mathcal{E} \wedge \mathcal{F}} \wedge\text{-intro} \right.$$

and

$$\pi' \left\{ \dfrac{\Gamma', u : \mathcal{E}, v : \mathcal{F} >> \mathcal{G}}{\Gamma' x : \mathcal{E} \wedge \mathcal{F} >> \mathcal{G}} \wedge\text{-elim} \right.$$

become

$$\dfrac{\dfrac{\Gamma >> \mathcal{F}}{\Delta >> \mathcal{F}} \quad \dfrac{\dfrac{\Gamma >> \mathcal{E}}{\Delta, v : \mathcal{F} >> \mathcal{E}} \quad \dfrac{\Gamma', u : \mathcal{E}, v : \mathcal{F} >> \mathcal{G}}{\Delta, u : \mathcal{E}, v : \mathcal{F} >> \mathcal{G}}}{\Delta, v : \mathcal{F} >> \mathcal{G}}\ \mathrm{seq}(\mathcal{E})}{\Delta >> \mathcal{G}}\ \mathrm{seq}(\mathcal{F})$$

$\mathcal{C} \equiv \mathcal{E} \to \mathcal{F}$

$$\pi \left\{ \frac{\Gamma, y : \mathcal{E} >> \mathcal{F}}{\Gamma >> \mathcal{E} \to \mathcal{F}} \to\text{-intro} \right.$$

and

$$\pi' \left\{ \frac{\Gamma' >> \mathcal{E} \qquad \Gamma', z : \mathcal{F} >> \mathcal{G}}{\Gamma', x : \mathcal{E} \to \mathcal{F} >> \mathcal{G}} \to\text{-elim} \right.$$

become

$$\dfrac{\dfrac{\Gamma' >> \mathcal{E}}{\Delta >> \mathcal{E}} \quad \dfrac{\Gamma, y : \mathcal{E} >> \mathcal{F}}{\Delta, y : \mathcal{E} >> \mathcal{F}}}{\Delta >> \mathcal{F}}\,\text{seq}(\mathcal{E}) \quad \dfrac{\Gamma', z : \mathcal{F} >> \mathcal{G}}{\Delta, z : \mathcal{F} >> \mathcal{G}}}{\Delta >> \mathcal{G}}\,\text{seq}(\mathcal{F})$$

$\mathcal{C} \equiv \mathcal{F}_1 \vee \mathcal{F}_2$

$$\pi \left\{ \frac{\Gamma >> \mathcal{F}_i}{\Gamma >> \mathcal{F}_1 \vee \mathcal{F}_2} \vee\text{-intro(left/right)} \right.$$

and

$$\pi' \left\{ \frac{\Gamma', u_1 : \mathcal{F}_1 >> \mathcal{G} \qquad \Gamma', u_2 : \mathcal{F}_2 >> \mathcal{G}}{\Gamma', x : \mathcal{F}_1 \vee \mathcal{F}_2 >> \mathcal{G}} \vee\text{-elim} \right.$$

become

$$\dfrac{\dfrac{\Gamma >> \mathcal{F}_i}{\Delta >> \mathcal{F}_i} \quad \dfrac{\Gamma', u_i : \mathcal{F}_i >> \mathcal{G}}{\Delta, u_i : \mathcal{F}_i >> \mathcal{G}}}{\Delta >> \mathcal{G}}\,\text{seq}(\mathcal{F}_i)$$

$\mathcal{C} \equiv \exists x : T.\mathcal{B}$

$$\pi \left\{ \frac{\Gamma >> \mathcal{B}[a/x]}{\Gamma >> \exists x : T.\mathcal{B}} \exists\text{-intro(a)} \right.$$

and

$$\pi' \left\{ \frac{\begin{array}{c}\vdots \pi'_1 \\ \Gamma', v : \mathcal{B}'[u/y] >> \mathcal{G}\end{array}}{\Gamma', z : \exists y : T.\mathcal{B}' >> \mathcal{G}} \exists\text{-elim(z,new[u,v]} \right.$$

become

$$\dfrac{\dfrac{\Gamma >> \mathcal{B}[a/x]}{\Delta >> \mathcal{B}[a/x]} \quad \dfrac{\begin{array}{c}\vdots \pi'_1[a/u] \\ \Gamma', v : \mathcal{B}'[a/y] >> \mathcal{G}\end{array}}{\Delta, v : \mathcal{B}'[a/y] >> \mathcal{G}}}{\Delta >> \mathcal{G}}\,\text{seq}(\mathcal{B}[a/x], \text{new[v]})$$

Note that we have allowed here for α-convertibility between $\mathcal{B}$ and $\mathcal{B}'[x/y]$, and that the substitution applied to the subproof π'_1 leaves the root sequent unchanged apart from where indicated by virtue of the eigen-condition on u.

C.6 Lemma 6: form of context

We show that the context of a formula $\mathcal{F}$ can be put in a form which is linear in size and membership of which can be checked in linear time (w.r.t. the size of $\mathcal{F}$).

The context of a formula is a set of sets of formulas. When fully enumerated the context is exponentially in size w.r.t. that of the formula. However, for a formula $\mathcal{F}$ containing no repeated propositional variables (which is the kind we consider), it can be shown that

$$\text{context}(\mathcal{F}) =< A, B >$$

where we define

$$< A, B >= \{A \cup x | x \in B\}$$

and s.t.

$$A \cap \bigcup B = \emptyset \tag{C.1}$$

$$B \text{ is pairwise disjoint} \tag{C.2}$$

In other words, each propositional variable appearing in a context appears either in every subset (those in A) or in exactly one (i.e. in one member of B). We also show a means of calculating the pair $< A, B >$ in time linear w.r.t. the size of $\mathcal{F}$.

C.6.1 Proof

Recall first that the function context is defined in the thesis as:

$$\begin{aligned} \texttt{context}([\cdot]) &= \{\emptyset\} \\ \texttt{context}(\mathcal{C}[\cdot]\#\mathcal{D}) &= \{\{\mathcal{D}\}\} \otimes \texttt{context}(\mathcal{C}[\cdot]) + \text{symmetric case} \\ \texttt{context}(\mathcal{C}[\cdot]|\mathcal{D}) &= \texttt{context}(\mathcal{C}[\cdot]) + \text{symmetric case} \end{aligned}$$

$$
\begin{aligned}
\texttt{context}(\mathcal{C} \rightarrow \mathcal{D}[\cdot]) &= \{\{\mathcal{C}\}\} \otimes \texttt{context}(\mathcal{D}[\cdot]) \\
\texttt{context}(\mathcal{C}[\cdot] \rightarrow \mathcal{D}) &= \texttt{context}(\mathcal{C}[\cdot]) \text{ where } \mathcal{D} \text{ is atomic} \\
\texttt{context}(\mathcal{C}[\cdot] \rightarrow \mathcal{P}\#\mathcal{Q}) &= \texttt{context}(\mathcal{C}[\cdot] \rightarrow \mathcal{P}) \cup \texttt{context}(\mathcal{C}[\cdot] \rightarrow \mathcal{Q}) \\
\texttt{context}(\mathcal{C}[\cdot] \rightarrow \mathcal{P}|\mathcal{Q}) &= \texttt{context}(\mathcal{C}[\cdot]) \\
\texttt{context}(\mathcal{C}[\cdot] \rightarrow \mathcal{P} \rightarrow \mathcal{Q}) &= \{\{\mathcal{P}\}\} \otimes \texttt{context}(\mathcal{C}[\cdot] \rightarrow \mathcal{Q})
\end{aligned}
$$

where

$$A \otimes B = \{x \cup y | x \in A \wedge y \in B\}$$

We now give a computationally oriented definition (`context'`)[2]:

$$
\begin{aligned}
\texttt{context'}([\cdot]) &= (\emptyset, \{\emptyset\}) \\
\texttt{context'}(\mathcal{C}[\cdot]\#\mathcal{D}) &= (\{\mathcal{D}\} \cup A, B) \text{ where } \texttt{context'}(\mathcal{C}[\cdot]) = (A, B) \\
&\quad + \text{symmetric case} \\
\texttt{context'}(\mathcal{C}[\cdot]|\mathcal{D}) &= \texttt{context'}(\mathcal{C}[\cdot]) + \text{symmetric case} \\
\texttt{context'}(\mathcal{C} \rightarrow \mathcal{D}[\cdot]) &= (\{\mathcal{C}\} \cup A, B) \text{ where } \texttt{context'}(\mathcal{D}[\cdot]) = (A, B) \\
\texttt{context'}(\mathcal{C}[\cdot] \rightarrow \mathcal{D}) &= \texttt{context'}(\mathcal{C}[\cdot]) \text{ where } \mathcal{D} \text{ is atomic} \\
\texttt{context'}(\mathcal{C}[\cdot] \rightarrow \mathcal{P}\#\mathcal{Q}) &= (A_1 \cup A_2, B_1 \cup B_2) \text{ where} \\
&\quad \texttt{context'}(\mathcal{C}[\cdot] \rightarrow \mathcal{P}) = (A_1, B_1) \text{ and} \\
&\quad \texttt{context'}(\mathcal{C}[\cdot] \rightarrow \mathcal{Q}) = (A_2, B_2) \\
\texttt{context'}(\mathcal{C}[\cdot] \rightarrow \mathcal{P}|\mathcal{Q}) &= \texttt{context'}(\mathcal{C}[\cdot]) \\
\texttt{context'}(\mathcal{C}[\cdot] \rightarrow \mathcal{P} \rightarrow \mathcal{Q}) &= (\{\mathcal{P}\} \cup A, B) \text{ where } \texttt{context'}(\mathcal{C}[\cdot] \rightarrow \mathcal{Q}) = (A, B)
\end{aligned}
$$

First we show the equivalence of these two definitions, i.e. that

$$\texttt{context'}(\mathcal{F}) = (A, B) \Longrightarrow \texttt{context}(\mathcal{F}) =< A, B >$$

Proof is by induction using the computational order used in both definitions.

---

[2]This algorithm clearly terminates by consideration of formula size in the first argument.

$\mathcal{F}[\cdot] = [\cdot]$

We have $< \emptyset, \{\emptyset\} >= \{\emptyset\}$ as required, with conditions (C.1) and (C.2) both satisified.

$\mathcal{F}[\cdot] = \mathcal{C}[\cdot] \# \mathcal{D}$

By IH we may suppose that

$$\texttt{context}(\mathcal{C}[\cdot]) =< A, B >$$

where $(A, B) = \texttt{context'}(\mathcal{C}[\cdot])$. Thus, it remains to show that

$$< \{\mathcal{D}\} \cup A, B >= \{\{\mathcal{D}\}\} \otimes < A, B >$$

But this is immediate from the definitions of $\otimes$ and $< \cdot, \cdot >$. Condition (C.2) is inherited from the IH, while condition (C.1) follows from the fact that the propositional variables of $\mathcal{D}$, and therefore $\mathcal{D}$ itself, do not appear in $\mathcal{C}[\cdot]$. [It seems too obvious to require proof here that all members of $\bigcup B$ are subformulas of $\mathcal{C}[\cdot]$.]

The symmetric case is proved similarly.

$\mathcal{F}[\cdot] = \mathcal{C}[\cdot] || \mathcal{D}$

We have the result required directly from the IH. Similarly for the symmetric case.

$\mathcal{F}[\cdot] = \mathcal{C} \rightarrow \mathcal{D}[\cdot]$

The proof is identical to that in section C.6.1.

$\mathcal{F}[\cdot] = \mathcal{C}[\cdot] \rightarrow \mathcal{D}$ **where** $\mathcal{D}$ **is atomic**

The proof is identical to that of section C.6.1.

$\mathcal{F}[\cdot] = \mathcal{C}[\cdot] \rightarrow \mathcal{P}\#\mathcal{Q}$

Let us suppose that $\texttt{context'}(\mathcal{C}[\cdot] \rightarrow \mathcal{P}) = (A_1, B_1)$ and $\texttt{context'}(\mathcal{C}[\cdot] \rightarrow \mathcal{Q}) = (A_2, B_2)$. Then by IH we have $\texttt{context}(\mathcal{C}[\cdot] \rightarrow \mathcal{P}) =< A_1, B_1 >$ and $\texttt{context}(\mathcal{C}[\cdot] \rightarrow \mathcal{Q}) =< A_2, B_2 >$, and must show that

$$< A_1, B_1 > \cup < A_2, B_2 > =< A_1 \cup A_2, B_1 \cup B_2 >$$

We do this by first showing that $B_1 = B_2$. In fact, if $\texttt{context'}(\mathcal{C}[\cdot]) = (A', B')$ for some A' and B', then for any formula $\mathcal{R}$, $\texttt{context'}(\mathcal{C}[\cdot] \rightarrow \mathcal{R}) = (A'', B')$ for some A''. Proof, which we omit here, is by induction on $\mathcal{R}$.

Hence, we have $B_1 = B_2 = B'$ for which condition (C.2) follows from the IH. And since the IHs now give us $A_1 \cap \bigcup B' = \emptyset$ and $A_2 \cap \bigcup B' = \emptyset$, condition (C.1) immediately follows.

$\mathcal{F}[\cdot] = \mathcal{C}[\cdot] \rightarrow \mathcal{P}|\mathcal{Q}$

The proof is identical to that of section C.6.1.

$\mathcal{F}[\cdot] = \mathcal{C}[\cdot] \rightarrow \mathcal{P} \rightarrow \mathcal{Q}$

The proof is identical to that section C.6.1.

C.6.2 Computation and complexity

The above proof provides an algorithm for caclulating the pair (A, B). From the computation order it can be seen that this algorithm has time complexity $O(s(\mathcal{F}))$, where $s(\mathcal{F})$ is the size of the input formula. From conditions (C.1) and (C.2) we can see that the output pair contains at most one occurrence of non-overlapping subformulas of $\mathcal{F}$, and thus that the algorithm also has space complexity $O(s(\mathcal{F}))$.

Now that we have shown that a context can be written in the form $< A, B >$ we also have a way to check membership in time $O(s(\mathcal{F}))$. To check whether $\mathcal{C} \in \texttt{context}(\mathcal{F}[\cdot])$, where $\texttt{context'}(\mathcal{F}[\cdot]) = (A, B)$, we :

1. Check whether $\mathcal{C} \in A$; if so, the answer is in the affirmative.

2. Check whether $\mathcal{C} \in D$ for each $D \in B$; if so, the answer is in the affirmative, otherwise the negative.

C.7 Lemma 7

This is a technical lemma used in other lemmas.

Let $\mathcal{D}$ be any formula in which no propositional variable is repeated, and Γ any valid hypothesis list s.t. all the free vars of $\mathcal{D}$ are declared there. Then, if none of the propositional vars of $\mathcal{D}$ occur positively in Γ and Γ is consistent,

$$\Gamma \not>> \mathcal{D}$$

C.7.1 Proof

Proof is by induction over a supposed proof of $\Gamma >> \mathcal{D}$.

hyp rule

This presupposes that $\mathcal{D} \in \Gamma$ and therefore that at least one prop. var. of $\mathcal{D}$ occurs positively in Γ. Contradiction.

thin rule

We have:

$$\frac{\Gamma' >> \mathcal{D}}{\Gamma >> \mathcal{D}} \text{thin(x)}$$

Since the IH applies to Γ' if it applies to Γ, we have

$$\Gamma' \not>> \mathcal{D}$$

by IH, and, therefore, a contradiction.

C.7.2 $\bot$-elim rule

This supposition contradicts the hypothesis that Γ is consistent.

C.7.3 $\wedge$-intro

We have:

$$\frac{\Gamma >> \mathcal{D}_1 \qquad \Gamma >> \mathcal{D}_2}{\Gamma >> \mathcal{D}_1 \# \mathcal{D}_2}$$

As in section C.7.1, the IH may be applied to either premise to yield a contradiction.

C.7.4 $\vee$-intro rule

We have:

$$\frac{\Gamma >> \mathcal{D}_i}{\Gamma >> \mathcal{D}_1 | \mathcal{D}_2}$$

As in section C.7.1, the IH may be applied to the premise to yield a contradiction.

C.7.5 $\rightarrow$-intro rule

We have:

$$\frac{\Gamma, \mathcal{D}_1 >> \mathcal{D}_2}{\Gamma >> \mathcal{D}_1 \rightarrow \mathcal{D}_2}$$

The IH can be applied to $\Gamma, \mathcal{D}_1 >> \mathcal{D}_2$, because of the condition on $\mathcal{D}$, to yield a contradiction. We additionally require that $\Gamma, \mathcal{D}_1$ be consistent. This follows simply from the interpolation theorem applied to $\Gamma, \mathcal{D}_1 >>$ `void`.

C.7.6 $\wedge$-elim rule

The IH applies equally to the premises after an $\wedge$-elim rule has been applied, so that we get a contradiction in the usual way.

C.7.7 ∨-elim rule

The IH applies equally to either of the premisses after an ∨-elim rule has been applied, so that we get a contradiction in the usual way.

C.7.8 →-elim rule

We have:

$$\frac{\Gamma, A \to B >> A \qquad \Gamma, A \to B, B >> \mathcal{D}}{\Gamma, A \to B >> \mathcal{D}} \to \text{-elim}$$

We may apply the IH to the right-hand premise for the same reasons as in section C.7.5 to yield a contradiction.

Q.E.D.

C.8 Lemma 8

This lemma deals with the case of disjunction in the main correctness theorem.

If

$$\Gamma, \mathcal{C}[K] >> \mathcal{C}[A] | \mathcal{D}$$

and the usual syntactic conditions apply to this sequent then

$$\Gamma, \mathcal{C}[K] >> \mathcal{C}[A]$$

Note that we may assume that the original formula has had boolean simplification applied so that `void` only occurs in subformulas of the form $\mathcal{P} \to$ `void`, i.e. negation. In paritcular, no subformula equivalent to `true` can occur.

C.8.1 Proof

Proof is by induction over the proof of $\Gamma, \mathcal{C}[K] >> \mathcal{C}[A] | \mathcal{D}$. We consider all paths going only through the RHS premise of →-elim rules and upto the first application

of $\vee$-intro (each such branch must have at least one such point, otherwise there is no possibility of applying the hyp rule to close the branch because of the conditions on $\mathcal{D}$). Let us call these nodes $\Delta_i >> \mathcal{C}[A]|\mathcal{D}$.

There are two cases. If the $\vee$-intro for node $\Delta_i >> \mathcal{C}[A]|\mathcal{D}$ is a right intro then we have $\Delta_i >> \mathcal{D}$, but this is impossible by lemma 7. Hence we must have left intros in every case, and for every node proofs of $\Delta_i >> \mathcal{C}[A]$. Now by replacing each node

$$\dfrac{\begin{array}{c}\vdots\,\pi\\ \Delta_i >> \mathcal{C}[A]\end{array}}{\Delta_i >> \mathcal{C}[A]|\mathcal{D}}\ \vee\text{-intro(left)}$$

with

$$\begin{array}{c}\vdots\,\pi\\ \Delta_i >> \mathcal{C}[A]\end{array}$$

we get a proof of $\Gamma >> \mathcal{C}[A]$.

C.9 Lemma 10

This is a technical lemma concerned with the $\rightarrow$ case of the main correctness theorem.

If

$$\mathcal{D} \rightarrow \mathcal{Q}, \Gamma, \mathcal{C}[K] \rightarrow \mathcal{D} >> \mathcal{D}$$

where $\mathcal{D}$ is atomic and the usual syntactic conditions apply, then

$$\Gamma, \mathcal{C}[K] \rightarrow \mathcal{D} >> \mathcal{D}$$

C.9.1 Proof

Proof is by induction over the proof of the first sequent.

hyp rule

From the syntactic conditions it is impossible that $\mathcal{D} \in \Gamma$.

thin rule

There are two cases. If $\mathcal{D} \rightarrow \mathcal{Q}$ is the thinned formula then we may use the proof of the premise as the proof of our goal. Otherwise we may apply the IH to the premise and the thin rule to the resulting proof.

$\bot$-elim rule

We must have $\bot \in \Gamma$, so that the $\bot$-elim rule can also be used to prove our goal.

intro rules

Since $\mathcal{D}$ is atomic there are no intro rules to consider.

$\wedge$-elim rule

We have

$$\frac{\mathcal{D} \rightarrow \mathcal{Q}, \Gamma', A\#B, \mathcal{C}[K] \rightarrow \mathcal{D}, A, B >> \mathcal{D}}{\mathcal{D} \rightarrow \mathcal{Q}, \Gamma', A\#B, \mathcal{C}[K] \rightarrow \mathcal{D} >> \mathcal{D}} \wedge\text{-elim}$$

where $\Gamma \equiv \Gamma', A\#B$. We may apply the IH to the premise and apply the $\wedge$-elim rule to the resulting proof to yield:

$$\frac{\Gamma', A\#B, \mathcal{C}[K] \rightarrow \mathcal{D}, A, B >> \mathcal{D}}{\Gamma', A\#B, \mathcal{C}[K] \rightarrow \mathcal{D} >> \mathcal{D}} \wedge\text{-elim}$$

$\vee$-elim rule

The proof in this case is analogous to that in the preceding section.

$\rightarrow$-elim rule

There are two cases to consider here. First, if the eliminated formula is $\mathcal{D} \rightarrow \mathcal{Q}$ then we have:

$$\frac{\mathcal{D} \rightarrow \mathcal{Q}, \Gamma, \mathcal{C}[K] \rightarrow \mathcal{D} >> \mathcal{D} \qquad \ldots}{\mathcal{D} \rightarrow \mathcal{Q}, \Gamma, \mathcal{C}[K] \rightarrow \mathcal{D} >> \mathcal{D}} \rightarrow\text{-elim}$$

We may apply the IH to the (identical) premise to yield our goal directly.

In the second case the eliminate formula is a member of Γ. In this case we may apply the IH to the RHS premise and reapply the $\rightarrow$-elim rule analagous to previous sections.

QED

C.9.2 Corollary

Iterating the above lemma to remove formulas of the form $\mathcal{D} \rightarrow \mathcal{Q}$ from Γ, and applying the interpolation theorem we get the corollary that:

$$\Gamma, \mathcal{C}[K] \rightarrow \mathcal{D} >> \mathcal{D}$$

where $\mathcal{D}$ is atomic and the usual syntactic conditions apply, implies

$$\Gamma >> \mathcal{D}$$

C.10 Lemma 11

This lemma shows that adding axioms at the sequent level is equivalent to adding them at the inference rule level.

Let $\mathcal{S}$ be a formula and Γ a valid hypothesis list s.t. all the free vars of $\mathcal{S}$ are declared in Γ. Now suppose we augment the proof system $>>$ by augmenting it with the axiom $\Gamma >> \mathcal{S}$. Let us denote this system $>>_{+\mathcal{S}}$. Then we have, for any formula $\mathcal{G}$:

$$\Gamma, \mathcal{S} >> G$$

iff

$$\Gamma >>_{+\mathcal{S}} \mathcal{G}$$

C.10.1 Proof

It is clear that any proof in $>>$ may be carried out unchanged in $>>_{+\mathcal{S}}$ so that, in particular, if we have $\Gamma, \mathcal{S} >> G$ then we have $\Gamma, \mathcal{S} >>_{+\mathcal{S}} G$ (indicated by the dotted line below).

$$\dfrac{\dfrac{}{\Gamma >>_{+\mathcal{S}} \mathcal{S}}\,\text{axiom} \qquad \begin{array}{c}\vdots\\ \Gamma, \mathcal{S} >>_{+\mathcal{S}} G\end{array}}{\Gamma >>_{+\mathcal{S}} \mathcal{G}}\,\text{seq}(\mathcal{S})$$

Appendix D

Reflection library

metaterm : complete
⊢ U1
Extract : rec(z.(atom|atom)|((atom#z)|(atom#(z#z))))

metaterm == term_of(metaterm)

tvar(v) == inl(inl(v))

tcon(c) == inl(inr(c))

tuna(f, a) == inr(inl(< f, a >))

tbin(f, a, b) == inr(inr(< f, < a, b >>))

metascheme : complete
⊢ $\forall phi$: metaterm → U1.
($\forall v$: atom.phi(tvar(v))) →
($\forall c$: atom.phi(tcon(c))) →
($\forall f$: atom.$\forall a$: metaterm.$phi(a)$ → phi(tuna(f, a))) →
($\forall f$: atom.$\forall a$: metaterm.$\forall b$: metaterm.$phi(a)$ → $phi(b)$ → phi(tbin(f, a, b))) →
($\forall t$: metaterm.$phi(t)$)

env : complete
⊢ U1
Extract : atom → pnat

env == term_of(env)

con_env == term_of(env)

una_env : complete
⊢ U1
Extract : atom → pnat → pnat

una_env == term_of(una_env)

bin_env : complete
⊢ U1
Extract : atom → pnat → pnat → pnat

bin_env == term_of(bin_env)

f_env : complete
⊢ U1
Extract : con_env#(una_env#bin_env)

f_env == term_of(f_env)

eval : complete
⊢ metaterm → env → f_env → pnat
Extract : $\lambda m.\lambda v0.\lambda v1.\mathtt{spread}(v1; v2, v3.\mathtt{spread}(v3; v5, v6.\ldots))$

$\mathtt{eval}(m, e, f) == \mathtt{term\_of}(\mathtt{eval})(m)(e)(f)$

zero_env : complete
⊢ env
Extract : λ_0

zero_env == term_of(zero_env)

zero_f_env : complete
⊢ f_env
Extract :$< \lambda\_0, < \lambda\_.\lambda\_0, \lambda\_.\lambda\_.\lambda\_0 >>$

zero_f_env == term_of(zero_f_env)

atom_decidable : complete
⊢ $\forall a$: atom.$\forall b$: atom.$a = b$ in atom $\vee$ ($\neg a = b$ in atom)

extend : complete
⊢ atom → pnat → env → env
Extract : $\lambda v.\lambda n.\lambda e.\lambda a.(\lambda v0.\mathtt{decide}(v0; \_.n; \_.e(a)))(\ldots)$

$\mathtt{extend}(v, n, e) == \mathtt{term\_of}(\mathtt{extend})(v)(n)(e)$

extend_con : complete
⊢ atom → pnat → f_env → f_env
Extract : $\lambda c.\lambda n.\lambda f.\mathtt{spread}(f; v0, v1. < \lambda a.(\lambda v3.\mathtt{decide}(v3; \_.n; \_.v0(a)))(\ldots), v1 >)$

$\mathtt{extend\_con}(c, n, f) == \mathtt{term\_of}(\mathtt{extend\_con})(c)(n)(f)$

extend_una : complete
⊢ atom → (pnat → pnat) → f_env → f_env
Extract : $\lambda f.\lambda k.\lambda e.\mathtt{spread}(e; v0, v1.\mathtt{spread}(v1; v3, v4. < v0, \ldots >))$

$\mathtt{extend\_una}(f, k, e) == \mathtt{term\_of}(\mathtt{extend\_una})(f)(k)(e)$

extend_bin : complete
⊢ atom → (pnat → pnat → pnat) → f_env → f_env
Extract : $\lambda f.\lambda k.\lambda e.\mathtt{spread}(e; v0, v1.\mathtt{spread}(v1; v3, v4. < v0, \ldots >))$

$\mathtt{extend\_bin}(f, k, e) == \mathtt{term\_of}(\mathtt{extend\_bin})(f)(k)(e)$

basic_f_env : complete
⊢ f_env
Extract :$< \lambda a.\mathtt{decide}(\mathtt{term\_of}(\mathtt{atom\_decidable})(a)(\text{"0"}); \_.0; \_.0), \ldots >$

basic == term_of(basic_f_env)

list_to_env : complete
⊢ (atom#pnat) list → env
Extract : $\lambda l.\lambda a.$list_ind$(l; 0; h, _, v.$spread$(h; v0, v1....))$

list_to_env(l) == term_of(list_to_env)(l)

rewrite : complete
⊢ U1
Extract : metaterm → metaterm

rewrite == term_of(rewrite)

simp : complete
⊢ f_env → U1
Extract : $\lambda f.$
$\{z :$ rewrite$|\forall m :$ metaterm.$\forall e : env.$eval$(m, e, f) =$ eval$(z(m), e, f)$ in pnat$\}$

simp(f) == term_of(simp)(f)

unroll_meta : complete
⊢ $\forall m :$ metaterm.
m in ((atom|atom)|((atom#metaterm)|(atom#(metaterm#metaterm))))

unroll_meta2 : complete
⊢ $\forall a :$ metaterm.$\forall b :$ metaterm.$a = b$ in metaterm →
$a = b$ in ((atom|atom)|((atom#metaterm)|(atom#(metaterm#metaterm))))

roll_meta : complete
⊢ $\forall m :$ (atom|atom)|((atom#metaterm)|(atom#(metaterm#metaterm))).
m in metaterm

metaterm_cases : complete
⊢ $\forall a :$ metaterm.
$(\exists f :$ atom.$a =$ tvar$(f) in$metaterm)$\vee$
$(\exists f :$ atom.$a =$ tcon(f) in metaterm)$\vee$
$(\exists f :$ atom.$\exists x :$ metaterm.$a =$ tuna(f, x) in metaterm)$\vee$
$(\exists f :$ atom.$\exists x :$ metaterm.$\exists y :$ metaterm.$a =$ tbin(f, x, y) in metaterm)

metaterm_decidable : complete
⊢ $\forall a :$ metaterm.$\forall b :$ metaterm.$a = b$ in metaterm $\vee (\neg a = b$ in metaterm)

reflect : complete
⊢ $\forall f :$ f_env.$\forall s :$ simp$(f).\forall t :$ pnat.$\forall m :$ metaterm.$\forall e : env.$
$t =$ eval(m, e, f) in pnat → $t =$ eval$(s(m), e, f)$ in pnat

basic_parts : complete
⊢ $\exists c :$ con_env.$\exists u :$ una_env.$\exists b :$ bin_env.basic $=< c, < u, b >>$ in f_env

left_destruct : complete
⊢ $\forall a :$ U1.$\forall b :$ U1.$\forall x : a.\forall y : a.inl(x) =$ inl(y) in $(a \vee b) \rightarrow x = y$ in a

right_destruct : complete
⊢ $\forall a :$ U1.$\forall b :$ U1.$\forall x : b.\forall y : b.inr(x) =$ inr(y) in $(a \vee b) \rightarrow x = y$ in b

union_contra : complete
$\vdash \forall a : \mathtt{U1}.\forall b : \mathtt{U1}.\forall x : a.\forall y : b.\neg \mathtt{inl}(x) = \mathtt{inr}(y)\ \mathtt{in}\ (a \vee b)$

union_contra_bis : complete
$\vdash \forall a : \mathtt{U1}.\forall b : \mathtt{U1}.\forall x : a.\forall y : b.\neg \mathtt{inr}(y) = \mathtt{inl}(x)\ \mathtt{in}\ (a \vee b)$

pair_destruct : complete
$\vdash \forall a : \mathtt{U1}.\forall b : \mathtt{U1}.\forall u : a.\forall v : a.\forall x : b.\forall y : b.$
$< u, x >=< v, y >\ \mathtt{in}\ (a \wedge b) \rightarrow (u = v\ \mathtt{in}\ a \wedge x = y\ \mathtt{in}\ b)$

var_wff : complete
$\vdash \forall v : \mathtt{atom.tvar}(v)\ \mathtt{in\ metaterm}$

con_wff : complete
$\vdash \forall c : \mathtt{atom.tcon}(c)\ \mathtt{in\ metaterm}$

una_wff : complete
$\vdash \forall f : \mathtt{atom}.\forall a : \mathtt{metaterm.tuna}(f, a)\ \mathtt{in\ metaterm}$

bin_wff : complete
$\vdash \forall f : \mathtt{atom}.\forall a : \mathtt{metaterm}.\forall b : \mathtt{metaterm.tbin}(f, a, b)\ \mathtt{in\ metaterm}$

partial : complete
$\vdash \mathtt{U1} \rightarrow \mathtt{U1}$
Extract : $\lambda t.(t \vee \mathtt{atom})$

$?(t) == \mathtt{term\_of(partial)}(t)$

$\mathtt{ok}(x) == \mathtt{inl}(x)$

ok_wff : complete
$\vdash \forall t : \mathtt{U1}.\forall x : t.\mathtt{ok}(x)\ \mathtt{in}\ ?(t)$

$\mathtt{failwith}(x) == \mathtt{inr}(x)$

failwith_wff : complete
$\vdash \forall t : \mathtt{U1}.\forall a : \mathtt{atom.failwith}(a)\ \mathtt{in}\ ?(t)$

$\mathtt{fail} == \mathtt{failwith}(" \sim ")$

decidable : complete
$\vdash \mathtt{U1} \rightarrow \mathtt{U1}$
Extract : $\lambda p.(p \vee (\neg p))$

$\delta(p) == \mathtt{term\_of}(decidable)(p)$

Appendix E

Context-sensitive source

```
/* context calculation routines */

/* test version */
context2(Term,Subterm,Context,Pol):-
        exp_at(Term,Pos,Subterm),
        context(Term,Pos,Context,Pol).

/* front end to context/5 below */
context(Term,Pos,Context,Pol):-
        reverse(Pos,RPos),
        context(RPos,Term,[],Context,+,Pol).

context_concl(H==>G,Pos,Context,Pol):-
    reverse(Pos,RPos),
    hyps_to_context(H,HH),
    context(RPos,G,HH,Context,+,Pol).

context_hyp(H==>G,Hyp,Pos,Context,Pol):-
    reverse(Pos,RPos),
    remove(Hyp:Type,H,HH),
    hyps_to_context(HH,HHH),
    context([1|RPos],Type=>G,HHH,Context,+,Pol).


/* context(+Pos,+Term,+ContextIn,-ContextOut,+Switch,-Polarity):
 * calculate the context for a position in a term. Switch is an
 * implementational detail. Note: contexts and positions are in
 * reverse order.
 */
context([2|Rest],C=>A,ConIn,ConOut,Switch,Pol):-        % implication
        split_prod(ConIn,C,NewCon),
        context(Rest,A,NewCon,ConOut,Switch,Pol).
context([1|Rest],A=>C,ConIn,ConOut,Switch,Pol):-        % special
        eager(C,ConIn,TempCon),                  % eager -> intros
```

```
        opposite(Switch,Op),
        context(Rest,A,TempCon,ConOut,Op,Pol).
context([1|Rest],A#C,ConIn,ConOut,Switch,Pol):- % conjunction
        split_prod(ConIn,C,TempCon),
        context(Rest,A,TempCon,ConOut,Switch,Pol).
context([2|Rest],C#A,ConIn,ConOut,Switch,Pol):-
        split_prod(ConIn,C,TempCon),
        context(Rest,A,TempCon,ConOut,Switch,Pol).
context([1|Rest],A\_C,ConIn,ConOut,Switch,Pol):-        % disjunction
        context(Rest,A,ConIn,ConOut,Switch,Pol).
context([2|Rest],_C\A,ConIn,ConOut,Switch,Pol):-
        context(Rest,A,ConIn,ConOut,Switch,Pol).
context([1|Rest],d(A),ConIn,ConOut,Switch,Pol):-        % decidability
        context(Rest,A,ConIn,ConOut,o,Pol).
context([2,2|Rest],V:C=>A,ConIn,ConOut,Switch,Pol):-    %universal
    member(V:_,ConIn)->
    hfree([NV],ConIn),
    s(A,[NV],[V],NA),
    context(Rest,NA,[NV:C|ConIn],ConOut,Switch,Pol);
    context(Rest,A,[V:C|ConIn],ConOut,Switch,Pol).
context([2,2|Rest],V:C#A,ConIn,ConOut,Switch,Pol):-     % existential
    member(V:_,ConIn)->
    hfree([NV],ConIn),
    s(A,[NV],[V],NA),
    context(Rest,NA,[NV:C|ConIn],ConOut,Switch,Pol);
    context(Rest,A,[V:C|ConIn],ConOut,Switch,Pol).
context([],_,Context,Context,Switch,Switch).            % base case


% switch polarities
opposite(+,-).
opposite(-,+).
opposite(o,o).

%remember: contexts are in reverse order
split_prod(ConIn,A#B,ConOut):-          %non-dependent
    split_prod(ConIn,A,Temp),
    split_prod(Temp,B,ConOut).
split_prod(ConIn,V:A#B,ConOut):-        %dependent
    member(V:_,ConIn)->
      hfree([NV],ConIn),
      s(B,[NV],[V],NB),
      split_prod([NV:A|ConIn],NB,ConOut);
      split_prod([V:A|ConIn],B,ConOut).
split_prod(ConIn,Term,[Term|ConIn]):-           %base case
    \+(Term=(_#_)),
```

```
    \+(Term=(_:_#_)).

%turn a standard hyp_list into a context
hyps_to_context(HypList,Context):-
    hyps_to_context(HypList,[],Context).
hyps_to_context([],ContextIn,ContextIn).
hyps_to_context([H:Type|T],ContextIn,ContextOut):-
    ((Type=(_#_);Type=(_:_#_))->
    split_prod(ContextIn,Type,NC);
    NC=[H:Type|ContextIn]
    ),
    hyps_to_context(T,NC,ContextOut).


%do as many -> intros as possible, splitting up any conjunctions
eager(A=>B,In,Out):-
        split_prod(In,A,Temp),
        eager(B,Temp,Out).
eager(C,In,In):-
        \+(C=(_=>_)).
```

Appendix F

Higher-order wave rule source

```
/* code for higher-order pattern unification.
 * See Miller and Nipkow.
 */
/* because we are using \\ and @ rather than lambda and of we need to
 * update Oyster's predicates dealing with syntax, e.g. freevars, s,
 * specifically:
 *
freevarinterm(V\\T,Var):-!,freevarinterm(T,Var),\+V=Var.
substituted(Var\\Pred, Insts, SubFrees, Bound, Avar\\Spred) :-
    !, substituted(Pred,[(Var-Avar)|Insts],[Avar|SubFrees],Bound,Spred).
substituted(Var\\Pred, Insts, SubFrees, Bound, Var\\Spred) :-
    substituted(Pred, Insts, SubFrees, [Var|Bound], Spred).
 *
 * See end.
 */
:-op(250,yfx,'@').              %meta application
:-op(250,xfy,'\\').             %meta abstraction

%auxilliaries
dummy_list(0,[]):-!.
dummy_list(N,[_|T]):-
        N>0,
        NN is N-1,
        dummy_list(NN,T).

%test whether Prolog variable Var in Term: already in Quintus library
/*
contains_var(Var,Term):-
        var(Var),
        \+(\+((numbervars(Term,0,_),nonvar(Var)))).
*/

beta_normal(X,X):-
        (var(X);\+functor(X,'\\',2),\+functor(X,@,2)),
```

```
        !.
beta_normal(A @ B,K):-
        beta_normal(A,AA),
        beta_normal(B,BB),
        (nonvar(AA),(AA=(X\\Z))->
                s(Z,[BB],[X],ZZ),
                beta_normal(ZZ,K);
                K=AA @ BB
        ).
beta_normal((X\\A),(X\\AA)):-
        beta_normal(A,AA).


eta_normal(X,X):-
        (var(X);\+functor(X,'\\',2),\+functor(X,@,2)),
        !.
eta_normal(A @ B,AA @ BB):-
        eta_normal(A,AA),
        eta_normal(B,BB).
eta_normal((X\\A),Z):-
        eta_normal(A,AA),
        ((nonvar(AA),AA=(AAA @ X),\+freevarinterm(AAA,X))->
                Z=AAA;
                Z=(X\\AA)
        ).

eta_normal_list([],[],_,_).
eta_normal_list([H|T],[HH|TT],Vars,Used):-
        eta_normal(H,HH),!,
        ((member(HH,Vars),\+member(HH,Used))->
            eta_normal_list(T,TT,Vars,[HH|Used]);
            write('unify: non bound var and distinct argument'),
            fail
        ).


/* pattern unification */
%top level
unify([]).
unify([(L=R)|T]):-
        beta_normal(L,LL),
        beta_normal(R,RR),
        unify1(LL,RR),
        unify(T).

unify1(L,R):-
        eta_expand(L,R,LL,RR),
```

```
        unify2(LL,RR).

unify2(L,R):-
        (
            triv_unify(L,R);
            triv_unify(R,L);
            rig_rig_unify(L,R);
            flex_rig_unify(L,R);
            flex_rig_unify(R,L);
            flex_flex_unify(L,R)
        ),
        !.

triv_unify(L,R):-
        destruct(L,Vars,Head,Args),
        var(Head),
        \+contains_var(Head,R), %occurs check
        Vars==Args,             %eta_expand ensures Args distinct b.v.'s
        Head=R.

rig_rig_unify(L,R):-
        destruct(L,Vars,LHead,LArgs),
        destruct(R,Vars,RHead,RArgs),
        ground(LHead),ground(RHead),
        LHead==RHead,                   %should really use convertible
        rig_rig_aux(LArgs,RArgs,Vars).

rig_rig_aux([],[],_):-!.
rig_rig_aux([S|SL],[U|UL],Vars):-
        !,
        abs(Vars,S,A),
        abs(Vars,U,B),
        unify1(A,B),            %note: need unify1 (eta_expand) here
        rig_rig_aux(SL,UL,Vars).
rig_rig_aux(_,_,_):-
        write('unify: untypable term'),
        fail.

flex_rig_unify(L,R):-
        destruct(L,Vars,LHead,LArgs),
        var(LHead),                             %flexible LHS
        is_pattern(LArgs,Vars,[]),              %check is a pattern
        destruct(R,Vars,RHead,RArgs),
        ground(RHead),                                  %rigid RHS
        \+((member(RHead,Vars),\+member(RHead,LArgs))), %ditto
        \+contains_var(LHead,RArgs),                    %occurs check
        flex_rig_aux(RArgs,LArgs,NewArgs),
```

```
        construct(LArgs,RHead,NewArgs,Z),
        unify1(LHead,Z),
        unify([L=R]).           %can we do better than use full unify

flex_rig_aux([],_,[]).
flex_rig_aux([_|T],Args,[A|Rest]):-
        curry(Args,_New,A),
        flex_rig_aux(T,Args,Rest).

is_pattern([],_,_):-!.
is_pattern([H|T],BVs,Used):-
        member(H,BVs),          %bound
        \+member(H,Used),       %and distinct
        !,
        is_pattern(T,BVs,[H|Used]).
is_pattern(_,_,_):-
        write('unify: not distinct bound vars'),
        fail.

flex_flex_unify(L,R):-
        destruct(L,Vars,LHead,LArgs),
        destruct(R,Vars,RHead,RArgs),
        var(LHead),var(RHead),
        ((LHead==RHead)->
            eta_normal_list(LArgs,Yn,Vars,[]),
            eta_normal_list(RArgs,Zn,Vars,[]),
            flex_flex_aux(Yn,Zn,Vp),
            construct(Yn,_New,Vp,Z),
            unify1(LHead,Z);
            eta_normal_list(LArgs,Yn,Vars,[]),
            eta_normal_list(RArgs,Zm,Vars,[]),
            flex_flex_aux2(Yn,Zm,Vp),
            construct(Yn,_New,Vp,Z1),
            unify1(LHead,Z1),
            construct(Zm,_New,Vp,Z2),
            unify([RHead=Z2])           %do we need full unify here?
        ).

flex_flex_aux([],[],[]):-!.
flex_flex_aux([H|T],[HH|TT],L):-
        !,
        (H==HH->
            L=[H|Rest];
            L=Rest
        ),
        flex_flex_aux(T,TT,Rest).
flex_flex_aux(_,_,_):-
```

```
        write('unify: untypable var'),
        fail.

flex_flex_aux2([],_,[]).
flex_flex_aux2([H|T],Vars,L):-
        (member(H,Vars)->
            L=[H|Rest];
            L=Rest
        ),
        flex_flex_aux2(T,Vars,Rest).


/* eta-expand L or R until they both have the same number of top-level
 * abstractions, and alpha-convert these so they are the same in each.
 */
eta_expand(L,R,NL,NR):-
        de_abs(L,LVars,LBody),
        de_abs(R,RVars,RBody),
        freevarsinterm(LBody,LFV),
        freevarsinterm(RBody,RFV),
        union([LVars,RVars,LFV,RFV],Vs),
        maplist(Vs,I:=>(I:_),true,Vars),
        eta_aux(LVars,RVars,LBody,RBody,LVars2,RVars2,
                LBody2,RBody2,Vars,New
               ),
        maplist(New,I:=>(I:_),true,NV),
        append(Vars,NV,NVs),
        maplist(LVars2,I:=>_,true,NVars),
        free(NVars,NVs),
        s(LBody2,NVars,LVars2,LBody3),
        s(RBody2,NVars,RVars2,RBody3),
        abs(NVars,LBody3,NL),
        abs(NVars,RBody3,NR).

eta_aux(LVars,RVars,LBody,RBody,LVars2,RVars2,LBody2,RBody2,Vars,New):-
        length(LVars,LL),
        length(RVars,RL),
        (LL<RL->
            Z is RL-LL,
            dummy_list(Z,New),
            free(New,Vars),
            curry(New,LBody,LBody2),
            append(LVars,New,LVars2),
            RVars=RVars2,
            RBody=RBody2;
        RL<LL->
            Z is LL-RL,
```

```
            dummy_list(Z,New),
            free(New,Vars),
            curry(New,RBody,RBody2),
            append(RVars,New,RVars2),
            LVars=LVars2,
            LBody=LBody2;
            New=[],
            LBody=LBody2,
            RBody=RBody2,
            LVars=LVars2,
            RVars=RVars2
        ).


/* predicates for conversion */
%split X into args and body. Use only de_abs(+,?,?).
de_abs(X,[],X):-
        (var(X);\+functor(X,'\\',2)),
        !.
de_abs((X\\A),[X|Vars],Body):-
        de_abs(A,Vars,Body).

%decurry a term as far as possible. Use only decurry(+,?,?).
decurry(Term,Head,Args):-
        decurry(Term,Head,[],Args).

%only decurry(+,?,?,?)
decurry(Term,Term,L,L):-
        (var(Term);\+functor(Term,@,2)),
        !.
decurry(A @ B,Head,In,Out):-
        decurry(A,Head,[B|In],Out).

%put the above two together
destruct(Term,Vars,Head,Body):-
        de_abs(Term,Vars,Rest),
        decurry(Rest,Head,Body).

%the constructive versions of the above, i.e. mode(+,+,-).
abs([],Body,Body).
abs([X|Vars],Body,(X\\A)):-
        abs(Vars,Body,A).


%curry(Args,TermIn,TermOut)
curry([],Head,Head).
```

```
curry([X|Args],TermIn,TermOut):-
        curry(Args,TermIn @ X,TermOut).

construct(Vars,Head,Body,Term):-
        curry(Body,Head,Temp),
        abs(Vars,Temp,Term).



/* testing some higher-order wave rules */
/* Note: to avoid variable capture on the righthand side we don't
 * allow logical vars to appear in the scope of an explicit lambda
 * binding. All this restriction can be messy all RHS's are normalised
 * after unification.
 */
ho_wave(wave @ F @ (wave @ G @ X),                %join
        wave @ ((f\\(g\\(x\\f @ (g @ x)))) @ F @ G) @ X
    ).
/*
ho_wave(plus @ (wave @ (x\\s @ (D @ x)) @ U) @ V,
        wave @ s @ (plus @ (wave @ D @ U) @ V)
    ).
ho_wave(even @ (wave @ (x\\s @ (s @ (D @ x))) @ U),
        even @ (wave @ D @ U)
    ).
ho_wave(eq @ (wave @ (x\\s @ (D1 @ x)) @ U)              %s cancellation
           @ (wave @ (x\\s @ (D2 @ x)) @ V),
        eq @ (wave @ D1 @ U) @ (wave @ D2 @ V)
    ).
ho_wave(member @ X @ (wave @ (x\\cons @ H @ (D @ x)) @ T),
        wave @ ((y\\(x\\ \ @ y @ x)) @ (eq @ X @ H)) @
                (member @ X @ (wave @ D @ T))
    ).
ho_wave(all @ T @ (x\\wave @ (z\\ \ @ (D @ z) @ Q) @ (P @ x)),
        wave @ ((q\\x\\ \ @ x @ q) @ Q)          %note P @ x
             @ (all @ T @ ((d\\p\\x\\wave @ d @ (p @ x)) @ D @ P))
    ).
ho_wave(map @ F @ (wave @ (x\\cons @ H @ (D @ x)) @ T),
        wave @ ((y\\(x\\cons @ y @ x)) @ (app @ F @ H))
             @ (map @ F @ (wave @ D @ T))
    ).
ho_wave(append @ (wave @ (x\\cons @ H @ (D @ x)) @ T) @ B,
        wave @ ((y\\(x\\cons @ y @ x)) @ H)
             @ (append @ (wave @ D @ T) @ B)
     ).
ho_wave(eq @ (wave @ (x\\cons @ H @ (C @ x)) @ A)        %cons cancellation
           @ (wave @ (x\\cons @ H @ (D @ x)) @ B),
```

```
        eq @ (wave @ C @ A) @ (wave @ D @ B)
    ).
ho_wave(app @ (wave @ (x\\comp @ F @ (D @ x)) @ G) @ X,
        wave @ ((y\\(x\\y @ x)) @ (app @ F)) @ (app @ (wave @ D @ G) @ X)
    ).
ho_wave(app @ (wave @ (x\\comp @ (D of x) @ G) @ F) @ X,        %sideways?
        app @ (wave @ D @ F) @ (wave @ ((y\\x\\app @ y @ x) @ G) @ X)
    ).
%multi-hole rules: see bbn 723
ho_wave(len @ (wave2 @ (x\\y\\append @ (C@x@y) @ (D@x@y)) @ L1 @ L2),
        wave2 @ (x\\y\\plus @ x @ y)
              @ (len @ (wave2 @ C @ L1 @ L2))
              @ (len @ (wave2 @ D @ L1 @ L2))
       ).
ho_wave(eq @ (wave2 @ (x\\y\\plus @ (C1@x@y) @ (D1@x@y)) @ U1 @ V1)
           @ (wave2 @ (x\\y\\plus @ (C2@x@y) @ (D2@x@y)) @ U2 @ V2),
        wave2 @ (x\\y\\ # @ x @ y)
              @ (eq @ (wave2 @ C1 @ U1 @ V1) @ (wave2 @ C2 @ U2 @ V2))
              @ (eq @ (wave2 @ D1 @ U1 @ V1) @ (wave2 @ D2 @ U2 @ V2))
          ).
ho_wave(eq @ (wave2 @ (x\\y\\plus @ (C1@x@y) @ (D1@x@y)) @ U1 @ V1)
           @ (wave2 @ (x\\y\\plus @ (C2@x@y) @ (D2@x@y)) @ V2 @ U2),
        wave2 @ (x\\y\\ # @ x @ y)
              @ (eq @ (wave2 @ C1 @ U1 @ V1) @ (wave2 @ D2 @ V2 @ U2))
              @ (eq @ (wave2 @ D1 @ U1 @ V1) @ (wave2 @ C2 @ V2 @ U2))
          ).
*/
ho_wave(eq @ (wave2 @ (x\\y\\plus @ (C1 @ x) @ (D1 @ y)) @ U1 @ V1)
           @ (wave2 @ (x\\y\\plus @ (C2 @ x) @ (D2 @ y)) @ U2 @ V2),
        wave2 @ (x\\y\\ # @ x @ y)
              @ (eq @ (wave @ C1 @ U1) @ (wave @ C2 @ U2))
              @ (eq @ (wave @ D1 @ V1) @ (wave @ D2 @ V2))
          ).
ho_wave(eq @ (wave2 @ (x\\y\\plus @ (C1 @ x) @ (D1 @ y)) @ U1 @ V1)
           @ (wave2 @ (x\\y\\plus @ (C2 @ y) @ (D2 @ x)) @ V2 @ U2),
        wave2 @ (x\\y\\ # @ x @ y)
              @ (eq @ (wave @ C1 @ U1) @ (wave @ D2 @ V2))
              @ (eq @ (wave @ D1 @ V1) @ (wave @ C2 @ U2))
          ).
/*
%miniscoping:
ho_wave(all @ T @ (x\\wave @ (z\\ # @ P @ (D @ z)) @ (Q @ x)),
        wave @ ((p\\z\\ # @ p @ z) @ P)
             @ (all @ T @ ((d\\q\\x\\wave @ d @ (q @ x)) @ D @ Q))
    ).
ho_wave(wave @ (x\\all @ T @ (z\\D @ x)) @ Q,   %quantifier elimination
        wave @ D @ Q
```

```
        ).
*/



%ripple(X,X):-
%       (X=(wave @ _ @ Y),fully_rippled(Y);fully_rippled(X)).
ripple(X,U):-
        sub_exp(X,Subterm,Pos),
        nonvar(Subterm),
        ho_wave(L,R),
        unify([L=Subterm]),
        beta_normal(R,RR),              %see remarks above ho_wave rules
        replace(Pos,RR,X,New),
        nl,wwrite(X),wwrite(' :=>'),nl,wwrite(New),nl,
        ripple(New,U).
ripple(X,X).

/* trial examples */
tterm(plus @ (wave @ C @ x)
          @ (wave @ D @ y),
      [C,D]).
tterm2(eq @ (plus @ (wave @ C @ x) @(wave @ D @ y))
         @ (wave @ E @ z),
       [C,D,E]
   ).
tterm3(eq @ (wave @ s @ x) @ (wave @ s @ y),
       []
   ).
tterm4(f @ (wave @ C @ x) @ (wave @ D @ y),
       [C,D]
   ).
tterm5(even @ (wave @ C @ x),[C]).
tterm6(even @ (plus @ (wave @ C @ x) @ (wave @ D @ y)),
       [C,D]
   ).
tterm7(plus @ (wave @ (x\\s @ (s @ (C @ x))) @ x)
           @ (wave @ D @ y),
       [C,D]
   ).
tterm8(plus @ (wave @ (x\\s @ (s @ x)) @ x) @ y,
       []
   ).
tterm9(even @ (wave @ (x\\s @ (C @ x)) @ x),
       [C]
   ).
```

```
tterm10(eq @ (plus @ (wave @ C @ x)
                     @ (plus @ (wave @ D @ y) @ (wave @ E @ z)))
           @ (plus @ (plus @ (wave @ C @ x) @ (wave @ D @ y))
                     @ (wave @ E @ z)),
        [C,D,E]
    ).
tterm11(member @ x @ (wave @ D @ l),[D]).
tterm12(all @ pnat @ (x\\wave @ (z\\ \ @ z @ q) @ p),[]).
tterm13(all @ pnat @ (x\\wave @ (z\\ \ @ z @ q) @ (p @ x)),[]).
tterm14(eq @ (map @ f @ (append @ (wave @ C @ x) @ (wave @ D @ y)))
           @ (append @ (map @ f @ (wave @ C @ x))
                     @ (map @ f @ (wave @ D @ y))),
        [C,D]
    ).
tterm15(eq @ (map @ (comp @ (wave @ C @ f) @ (wave @ D @ g))
                  @ (wave @ E @ l))
           @ (map @ (wave @ C @ f)
                  @ (map @ (wave @ D @ g) @ (wave @ E @ l))),
        [C,D,E]
    ).
tterm16(comp @ (wave @ C @ f) @ (wave @ D @ g) @ x,[C,D]).
tterm17(eq @ (len @ (wave2 @ (x\\y\\append @ x @ y) @ a @ b))
           @ (len @ (wave2 @ (x\\y\\append @ y @ x) @ a @ b)),
        []
    ).
tterm18(len @ (wave2 @ (x\\y\\append @ x @ y) @ a @ b),
        []
    ).
tterm19(eq @ (wave2 @ (x\\y\\plus @ x @ y) @ a @ b)
           @ (wave2 @ (x\\y\\plus @ y @ x) @ a @ b),
        []
    ).
tterm20(eq @ (wave2 @ (x\\y\\plus @ y @ x) @ b @ a)
           @ (wave2 @ (x\\y\\plus @ x @ y) @ b @ a),
        []
    ).
tterm21(all @ pnat @ (x\\wave @ (z\\ # @ p @ (neg @ z)) @ (q @ x)),
        []
    ).
tterm22(all @ pnat @ (x\\wave @ (z\\ \ @ (neg @ z) @ q) @ (p @ x)),
        []
    ).
tterm23(wave @ (x\\all @ pnat @ (z\\neg @ x)) @ p,
        []
        ).
tterm100(even('@wave_front@'(hard,out,s(s('@wave_var@'(s(0)))))),
        []
```

```
    ).

wwrite(X):-
    convert_down(X,XX),
    print(XX).

fully_rippled(Term):-
        \+((
        exp_at(Term,Pos,(wave @ G @ Y)),
        \+((eta_normal(G,GG),(var(GG);GG=(X\\X))))
    )).


%best-first subexpressions (try and recode so tail-recursive).
sub_exp(Term,Subterm,Pos):-
        sub_exp_aux(Term,Subterm,Pos,0).

sub_exp_aux(Term,Subterm,Pos,Depth):-
        sub_exp(Term,_,_,Depth)->
        (sub_exp(Term,Subterm,Pos,Depth);
        sub_exp_aux(Term,Subterm,Pos,s(Depth))).

sub_exp(A,A,[],0).
sub_exp(Term,Subterm,Pos,s(Depth)):-
        nonvar(Term),functor(Term,@,2),
        decurry(Term,Head,Args),
        reverse(Args,RArgs),
        add_pos(RArgs,PArgs,[]),
        re_order(PArgs,NArgs),
        member(Suffix-Sub,NArgs),
        sub_exp(Sub,Subterm,PPos,Depth),
        append(PPos,Suffix,Pos).

add_pos([],[],_).
add_pos([H|T],[[2|Pos]-H|TT],Pos):-
        add_pos(T,TT,[1|Pos]).

re_order(X,X).


/* routines for conversion between object and abstract syntax */
/* convert concrete to asbtract */
convert_up(Var,Var):-
        var(Var),!.
convert_up({Name},Name):-
        atom(Name).
convert_up(Var:Type#Pred,some @ T @ (Var\\P)):-
```

```
        !,
        convert_up(Type,T),
        convert_up(Pred,P).
convert_up(Var:Type=>Pred,all @ T @ (Var\\P)):-
        !,
        convert_up(Type,T),
        convert_up(Pred,P).
convert_up({Var:Type\Pred},set @ T @ (Var\\P)):-
        !,
        convert_up(Type,T),
        convert_up(Pred,P).
convert_up(lambda(Var,Body),lambda @ (Var\\B)):-
        !,
        convert_up(Body,B).
convert_up([H|T],LL):-
        !,append(BVs,[Body],[H|T]),
        convert_up(Body,B),
        abs(BVs,B,LL).
convert_up('@wave_front@'(_Type,_Dir,Front),wave @ NFront @ H):-
        !,
        genvar(V),
        \+(appears(V,Front)),!,
        wave_fronts(FF,[[]-[Pos]/[_,_]],'@wave_front@'(_Type,_Dir,Front)),
%i.e. single hole only at present
        exp_at(FF,Pos,Hole),
        replace(Pos,V,FF,K),
        convert_up(K,F),
        eta_normal(V\\F,NFront),
        convert_up(Hole,H).
convert_up(Term,T):-
        Term=..[F|Args],
        maplist(Args,I:=>O,convert_up(I,O),NArgs),
        curry(NArgs,F,T).

/* convert abstract to concrete: input should be normalised */
convert_down(Var,Var):-
        var(Var),!.
convert_down(some @ T @ (Var\\P),Var:TT#PP):-
        !,
        convert_down(T,TT),
        convert_down(P,PP).
convert_down(all @ T @ (Var\\P),Var:TT=>PP):-
        !,
        convert_down(T,TT),
        convert_down(P,PP).
convert_down(set @ T @ (Var\\P),{Var:TT\PP}):-
        !,
```

```
        convert_down(T,TT),
        convert_down(P,PP).
convert_down(lambda @ (Var\\Body),lambda(Var,B)):-
        !,
        convert_down(Body,B).
convert_down(V\\B,L):-
        !,
        de_abs(V\\B,Vars,Body),
        convert_down(Body,BB),
        append(Vars,[BB],L).
convert_down(wave @ Front @ Hole,'@wave_front@'(hard,out,R)):-
        !,
        convert_down(Hole,H),
        beta_normal(Front @ _Hole,RR),
        convert_down(RR,R),
        _Hole='@wave_var@'(H).
convert_down(Term,T):-
        decurry(Term,Head,Args),
        maplist(Args,I:=>O,convert_down(I,O),NArgs),
        T=..[Head|NArgs].



/* stuff needed to update Oyster's substituted/5 and freevarinterm/2 */

%existing Oyster code
freevarinterm(X, _) :- var(X), !, fail.
freevarinterm(term_of(_),_) :- !,fail.
freevarinterm(atom(_),_)  :- !,fail.
freevarinterm({N},_) :- atom(N),!,fail.
freevarinterm([H|T], Var) :-
    !, append(BoundVars, [Term], [H|T]), freevarinterm(Term, Var),
    \+ member(Var, BoundVars).
freevarinterm((_:T1#_), Var) :- freevarinterm(T1,Var).
freevarinterm((V:_#T2), Var) :- !, freevarinterm(T2,Var), \+ Var = V.
freevarinterm((_:T1=>_), Var) :- freevarinterm(T1,Var).
freevarinterm((V:_=>T2), Var) :- !, freevarinterm(T2,Var), \+ Var = V.
freevarinterm(({_:T1\_}), Var) :- freevarinterm(T1,Var).
freevarinterm(({V:_\T2}), Var) :- !, freevarinterm(T2,Var), \+ Var = V.
%new piece
freevarinterm(V\\T,Var):-!,freevarinterm(T,Var),\+V=Var.
%existing Oyster code
freevarinterm(lambda(V,T), Var) :- !, freevarinterm(T, Var), \+ V = Var.
freevarinterm(Var,Var) :- ttvar(Var).
freevarinterm(Tm, Var) :-
        Tm =.. [_|Args], member(Arg,Args), freevarinterm(Arg, Var).
```

```
%existing Oyster code
substituted(Var,_,_,_,Var) :- var(Var),!.
substituted(Term,  _ , _ , Bound, Term) :- member(Term,Bound),!.
substituted(Term, Insts, _ , Bound, Instd) :-
    member((Term - Instd), Insts), !, \+ member(Term, Bound).
substituted(su(Term,New,Old),Insts,Subfrees,Bound,SS) :-
    s(Term,New,Old,TT),substituted(TT,Insts,Subfrees,Bound,SS).
substituted(atom(Name),_,_,_,atom(Name)).
substituted(term_of(Name),_,_,_,term_of(Name)).
substituted({Name},_,_,_,{Name}) :- atom(Name).
substituted({Var:Type\Pred}, Insts, SubFrees, Bound, {Avar:Stype\Spred}):-
    member(Var, SubFrees),
    substituted(Type, Insts, SubFrees, Bound, Stype),
    modify(Var, Avar), \+ member(Avar, SubFrees), !,
    substituted(Pred, [(Var-Avar)|Insts], [Avar|SubFrees], Bound, Spred).
substituted({Var:Type\Pred}, Insts, SubFrees, Bound, {Var:Stype\Spred}) :-
    substituted(Type, Insts, SubFrees, Bound, Stype),
    substituted(Pred, Insts, SubFrees, [Var|Bound], Spred).
substituted((Var:Type#Pred), Insts, SubFrees, Bound, (Avar:Stype#Spred)):-
    member(Var, SubFrees),
    substituted(Type, Insts, SubFrees, Bound, Stype),
    modify(Var, Avar), \+ member(Avar, SubFrees),
    !,
    substituted(Pred, [(Var-Avar)|Insts], [Avar|SubFrees], Bound, Spred).
substituted((Var:Type#Pred), Insts, SubFrees, Bound, (Var:Stype#Spred)) :-
    substituted(Type, Insts, SubFrees, Bound, Stype),
    substituted(Pred, Insts, SubFrees, [Var|Bound], Spred).
substituted((Var:Type=>Pred),Insts,SubFrees,Bound,(Avar:Stype=>Spred)):-
    member(Var, SubFrees),!,
    substituted(Type, Insts, SubFrees, Bound, Stype),
    modify(Var, Avar), \+ member(Avar, SubFrees),!,
    substituted(Pred, [(Var-Avar)|Insts], [Avar|SubFrees], Bound, Spred).
substituted((Var:Type=>Pred),Insts,SubFrees,Bound,(Var:Stype=>Spred)):-
    !,substituted(Type, Insts, SubFrees, Bound, Stype),
    substituted(Pred, Insts, SubFrees, [Var|Bound], Spred).
%new piece
substituted(Var\\Pred, Insts, SubFrees, Bound, Avar\\Spred) :-
    member(Var, SubFrees), modify(Var, Avar), \+ member(Avar, SubFrees),
    !,
    substituted(Pred, [(Var-Avar)|Insts], [Avar|SubFrees], Bound, Spred).
substituted(Var\\Pred, Insts, SubFrees, Bound, Var\\Spred) :-
    substituted(Pred, Insts, SubFrees, [Var|Bound], Spred).
%existing Oyster code
substituted(lambda(Var,Pred),Insts,SubFrees,Bound,lambda(Avar,Spred)):-
    member(Var, SubFrees), modify(Var, Avar), \+ member(Avar, SubFrees),
    !,
```

```
    substituted(Pred, [(Var-Avar)|Insts], [Avar|SubFrees], Bound, Spred).
substituted(lambda(Var,Pred),Insts,SubFrees,Bound,lambda(Var,Spred)):-
    substituted(Pred, Insts, SubFrees, [Var|Bound], Spred).
substituted(rec(Var,Pred), Insts, SubFrees, Bound, rec(Avar,Spred)) :-
    member(Var, SubFrees), modify(Var, Avar), \+ member(Avar, SubFrees),
    !,
    substituted(Pred, [(Var-Avar)|Insts], [Avar|SubFrees], Bound, Spred).
substituted(rec(Var,Pred), Insts, SubFrees, Bound, rec(Var,Spred)) :-
    substituted(Pred, Insts, SubFrees, [Var|Bound], Spred).
substituted([~,Binding], Insts, SubFrees, Bound, [~,SBinding]) :-
    !, substituted(Binding, Insts, SubFrees, Bound, SBinding).
substituted([~|Binding], Insts, SubFrees, Bound, [~|SBinding]) :-
    !, substituted(Binding, Insts, SubFrees, Bound, SBinding).
substituted([Var,Pred], Insts, SubFrees, Bound, [Avar,Spred]) :-
    member(Var, SubFrees),
    modify(Var, Avar),
    \+ member(Avar, SubFrees),
    !,
    substituted(Pred, [(Var-Avar)|Insts], [Avar|SubFrees], Bound, Spred).
substituted([Var,Pred], Insts, SubFrees, Bound, [Var,Spred]) :-
    !, substituted(Pred, Insts, SubFrees, [Var|Bound], Spred).
substituted([Var|Bind], Insts, SubFrees, Bound, [Avar|SBind]) :-
    member(Var, SubFrees), modify(Var, Avar), \+ member(Avar, SubFrees),
    !,
    substituted(Bind, [(Var-Avar)|Insts], [Avar|SubFrees], Bound, SBind).
substituted([Var|Bind], Insts, SubFrees, Bound, [Var|SBind]) :-
    !, substituted(Bind, Insts, SubFrees, [Var|Bound], SBind).
substituted(Term, _, _, _, Term) :- atomic(Term), !.
substituted(Term, Insts, SubFrees, Bound, STerm) :-
    \+ var(Term),
    Term =.. [Funct|Args],
    \+member(Funct, [su,atom,term_of,{},:,'\\',lambda,rec,~]),  %note '\\'
    !,
    substitutedlist(Args, Insts, SubFrees, Bound, SArgs),
    STerm =.. [Funct|SArgs].
substituted(Term, Insts, SubFrees, Bound, STerm) :-
    \+ var(STerm),
    STerm =.. [SFunct|SArgs],
    \+member(SFunct, [su,atom,term_of,{},:,'\\',lambda,rec,~]), %note '\\'
    substitutedlist(Args, Insts, SubFrees, Bound, SArgs),
    Term =.. [SFunct|Args].
```

Appendix G

strong_fertilization method

```
submethod(strong_fertilize([Hyp,[]]),
          H==>G,
          [hyp(Hyp:Hypothesis,H),
           skolemize(G,[],_AV,[],EV,Inst),
           co_skolemize(Hypothesis,[],V,Pat),
           unify(Pat,Inst)
          ],
          [],
          [],
          strong_fertilize([Hyp,[]])
         ).

submethod(strong_fertilize([Hyp,WHPos]),
          H==>G,
          [hyp(Hyp:Hypothesis,H),
           skolemize(G,[],_AV,[],EV,Matrix),
           co_skolemize(Hypothesis,[],V,Pat),
           wave_fronts(Mat,WFs,Matrix),
           del(WFs,([]-WHPosList/[Typ,Dir]),RestWFs),
           del(WHPosList,WHPos,RestWHs),
           exp_at(Matrix,WHPos,SubMatrix),
           wave_fronts(_,[],SubMatrix),
           unify(Pat,SubMatrix)
          ],
          [replace(WHPos,{true},Mat,NewMat),
           (RestWHs=[]->wave_fronts(NewMat,RestWFs,NewMatrix);
            wave_fronts(NewMat,[[]-RestWHs/[Typ,Dir]|RestWFs],NewMatrix)
           ),
           de_skolemize(G,NewMatrix,EV,NewG)
          ],
          [H==>NewG],
          strong_fertilize([Hyp,WHPos])
      ).
```

Appendix H

Extract terms

$$
\begin{array}{lll}
\lambda i.\texttt{tvar}(u) \mapsto \lambda pos, x, n, hyp. < n, \_ > & & \\
\quad \texttt{tcon}(u) \mapsto \lambda pos, x, n, hyp. < n, \_ > & & \\
\quad \texttt{tuna}(f, a), h1 \mapsto \lambda pos. & \texttt{nil} \mapsto \lambda x, n, hyp. < n, \_ > & \\
 & h :: t, h3 \mapsto \lambda x, n, hyp. < \texttt{tuna}(f, p2(h1txn\_), \_ > & \\
\quad \texttt{tbin}(f, a, b), h1, h2 \mapsto \lambda pos. & \texttt{nil} \mapsto \lambda x, n, hyp. & \\
 & h = 0 \mapsto < \texttt{tbin}(f, p2(h1txn\_), b), \_ > & \\
 & h = 1 \mapsto < \texttt{tbin}(f, a, p2(h2txn\_)), \_ > &
\end{array}
$$

$$
\begin{array}{lll}
\lambda i.\texttt{tvar}(v) \mapsto \lambda pos, x, n. & \delta(pos = \texttt{nil in posn}) \texttt{ AND} & \\
 & \delta(x = \texttt{tvar}(v) \texttt{ in metaterm}) \texttt{ AND} & \\
 & \delta(\texttt{def\_eqn}(x, n)) & \\
\quad \texttt{tcon}(v) \mapsto \lambda pos, x, n. & \delta(pos = \texttt{nil in posn}) \texttt{ AND} & \\
 & \delta(x = \texttt{tcon}(v) \texttt{ in metaterm}) \texttt{ AND} & \\
 & \delta(\texttt{def\_eqn}(x, n)) & \\
\quad \texttt{tuna}(f, a), h1 \mapsto \lambda pos. & \texttt{nil} \mapsto \lambda x, n. & \delta(\texttt{tuna}(f, a) = x) \texttt{ AND} \\
 & & \delta(\texttt{def\_eqn}(x, n)) \\
 & h :: t, h3 \mapsto \lambda x, n. & \delta(h = 0 \texttt{ in pnat}) \texttt{ AND} \\
 & & (h1xn) \texttt{ AND} \\
 & & \delta(\texttt{def\_eqn}(x, n)) \\
\quad \texttt{tbin}(f, a, b), h1, h2 \mapsto \lambda pos. & \texttt{nil} \mapsto \lambda x, n. & \delta(\texttt{tbin}(f, a, b) = x \texttt{ in metaterm}) \texttt{ AND} \\
 & & \delta(\texttt{def\_eqn}(x, n)) \\
 & h :: t, h3 \mapsto \lambda x, n. & (\delta(h = 0 \texttt{ in pnat}) \texttt{ AND} \\
 & & (h1xn)) \texttt{ OR} \\
 & & (\delta(h = s(0) \texttt{ in pnat}) \texttt{ AND} \\
 & & (h2xn))
\end{array}
$$

Appendix I

Program 1

```
prog(tvar(V),Pos,X,N,O):-N=O.
prog(tcon(V),Pos,X,N,O):-N=O.
prog(tuna(F,A),Pos,X,N,O):-
        aux1(Pos,F,A,X,N,O).
prog(tbin(F,A,B),Pos,X,N,O):-
        aux2(Pos,F,A,B,X,N,O).

aux1(nil,F,A,X,N,O):-N=O.
aux1(H::T,F,A,X,N,O):-
        prog(A,T,X,N,Z),
        O=tuna(F,Z).

aux2(nil,F,A,B,X,N,O):-N=O.
aux2(H::T,F,A,B,X,N,O):-
        aux3(H,T,F,A,B,X,N,O).

aux3(H,T,F,A,B,X,N,O):-
        H=0,
        effects(A,T,X,N,Z),
        O=tbin(F,Z,B).
aux3(H,T,F,A,B,X,N,O):-
        H=s(0),
        effects(B,T,X,N,Z),
        O=tbin(F,A,Z).
```

Appendix J

Program 2

```
pred(tvar(V),Pos,X,N):-
        Pos=nil, X=tvar(V), def_eqn_pred(X,N).
pred(tcon(V),Pos,X,N):-
        Pos=nil, X=tcon(V), def_eqn_pred(X,N).
pred(tuna(F,A),Pos,X,N):-
        aux1(Pos,F,A,X,N).
pred(tbin(F,A,B),Pos,X,N):-
        aux2(Pos,F,A,B,X,N).

aux1(nil,F,A,X,N):-
        X=tuna(F,A),
        def_eqn_pred(X,N).
aux1(H::T,F,A,X,N):-
        H=0,
        pred(A,T,X,N),
        def_eqn_pred(X,N).

aux2(nil,F,A,B,X,N):-
        X=tbin(F,A,B),
        def_eqn_pred(X,N).
aux2(H::T,F,A,B,X,N):-
        H=0,pred(A,T,X,N),def_eqn_pred(X,N);
        H=s(0),pred(B,T,X,N),def_eqn_pred(X,N).
```

Appendix K

Glossary

$i \sim o$

$$\texttt{tvar}(u) \sim \texttt{tvar}(v) \leftrightarrow u = v \texttt{ in atom}$$
$$\texttt{tcon}(u) \sim \texttt{tcon}(v) \leftrightarrow u = v \texttt{ in atom}$$
$$\boxed{\texttt{tuna}(f, \underline{a})} \sim \boxed{\texttt{tuna}(g, \underline{c})} \leftrightarrow \boxed{(f = g \texttt{ in atom}) \wedge \underline{a \sim c}}$$
$$\boxed{\texttt{tbin}(f, \underline{a_1}, \underline{b_2})} \sim \boxed{\texttt{tbin}(g, \underline{c_1}, \underline{d_2})} \leftrightarrow \boxed{(f = g \texttt{ in atom}) \wedge \underline{a \sim c_1} \wedge \underline{b \sim d_2}}$$

$\texttt{exp\_at}(i : \texttt{metaterm}, pos : \texttt{posn}, sub : \texttt{metaterm})$

...succeeds if *sub* is the subterm at position *pos* in i.

$$\texttt{exp\_at}(\texttt{tvar}(v), pos, x) \leftrightarrow pos = \texttt{nil in posn} \wedge x = \texttt{tvar}(v) \texttt{ in metaterm}$$
$$\texttt{exp\_at}(\texttt{tcon}(v), pos, x) \leftrightarrow pos = \texttt{nil in posn} \wedge x = \texttt{tcon}(v) \texttt{ in metaterm}$$
$$\texttt{exp\_at}(\texttt{tuna}(f, a), \texttt{nil}, x) \leftrightarrow x = \texttt{tuna}(f, a) \texttt{ in metaterm}$$
$$\texttt{exp\_at}(\boxed{\texttt{tuna}(f, \underline{a})}, \boxed{h :: \underline{t}}, x) \leftrightarrow \boxed{(h = 0 \texttt{ in pnat}) \wedge \underline{\texttt{exp\_at}(a, t, x)}}$$
$$\texttt{exp\_at}(\texttt{tbin}(f, a, b), \texttt{nil}, x) \leftrightarrow x = \texttt{tbin}(f, a, b) \texttt{ in metaterm}$$
$$\texttt{exp\_at}(\boxed{\texttt{tbin}(f, \underline{a_1}, \underline{b_2})}, \boxed{h :: \underline{t_{1,2}}}, x) \leftrightarrow$$
$$\boxed{(h = 0 \texttt{ in pnat} \wedge \underline{\texttt{exp\_at}(a, t, x)_1}) \vee (h = 1 \texttt{ in pnat} \wedge \underline{\texttt{exp\_at}(b, t, x))_2}}$$

$\mathtt{replace}(i : \mathtt{metaterm}, pos : \mathtt{posn}, s : \mathtt{metaterm}, n : \mathtt{metaterm})$

...succeeds if n is the result of replacing the subterm at position pos in i with s.

$\mathtt{replace}(\mathtt{tvar}(v), pos, s, n) \leftrightarrow pos = \mathtt{nil}\ \mathtt{in}\ posn \wedge s = n\ \mathtt{in}\ \mathtt{metaterm}$
$\mathtt{replace}(\mathtt{tcon}(v), pos, s, n) \leftrightarrow pos = \mathtt{nil}\ \mathtt{in}\ posn \wedge s = n\ \mathtt{in}\ \mathtt{metaterm}$
$\mathtt{replace}(\mathtt{tuna}(f, a), \mathtt{nil}, s, n) \leftrightarrow s = n\ \mathtt{in}\ \mathtt{metaterm}$
$\mathtt{replace}(\mathtt{tuna}(f, a), h::t, s, \mathtt{tvar}(v)) \leftrightarrow \mathtt{void}$
$\mathtt{replace}(\mathtt{tuna}(f, a), h::t, s, \mathtt{tcon}(v)) \leftrightarrow \mathtt{void}$
$\mathtt{replace}(\boxed{\mathtt{tuna}(f, \underline{a})}, \boxed{h::\underline{t}}, s, \boxed{\mathtt{tuna}(g, \underline{c})}) \leftrightarrow$
$\boxed{f = g\ \mathtt{in}\ \mathtt{atom} \wedge h = 0\ \mathtt{in}\ \mathtt{pnat} \wedge \underline{\mathtt{replace}(a, t, s, c)}}$
$\mathtt{replace}(\mathtt{tuna}(f, a), h::t, s, \mathtt{tbin}(g, c, d)) \leftrightarrow \mathtt{void}$
$\mathtt{replace}(\mathtt{tbin}(f, a, b), \mathtt{nil}, s, n) \leftrightarrow s = n\ \mathtt{in}\ \mathtt{metaterm}$
$\mathtt{replace}(\mathtt{tbin}(f, a, b), h::t, s, \mathtt{tvar}(v)) \leftrightarrow \mathtt{void}$
$\mathtt{replace}(\mathtt{tbin}(f, a, b), h::t, s, \mathtt{tcon}(v)) \leftrightarrow \mathtt{void}$
$\mathtt{replace}(\mathtt{tbin}(f, a, b), h::t, s, \mathtt{tuna}(g, c)) \leftrightarrow \mathtt{void}$
$\mathtt{replace}(\boxed{\mathtt{tbin}(f, \underline{a_1}, \underline{b_2})}, \boxed{h::\underline{t_{1,2}}}, s, \boxed{\mathtt{tbin}(g, \underline{c_1}, \underline{d_2})}) \leftrightarrow$
$\boxed{f = g\ \mathtt{in}\ \mathtt{atom} \wedge ((h = 0\ \mathtt{in}\ \mathtt{pnat} \wedge b = d\ \mathtt{in}\ \mathtt{metaterm} \wedge \underline{\mathtt{replace}(a, t, s, c)}_1) \vee$
$\boxed{(h = 1\ \mathtt{in}\ \mathtt{pnat} \wedge a = c\ \mathtt{in}\ \mathtt{metaterm} \wedge \underline{\mathtt{replace}(b, t, s, d)}_2))}$

$\mathtt{wave\_rule}(l, r)$

$\forall l, r : \mathtt{metaterm}.\mathtt{wave\_rule}(l, r) \rightarrow l \sim r$
$\forall l, r : \mathtt{metaterm}.\delta(\mathtt{wave\_rule}(l, r))$

$\mathtt{def\_eqn}(l, r)$

$\forall l, r : \mathtt{metaterm}.\mathtt{def\_eqn}(l, r) \rightarrow l \sim r$
$\forall l, r : \mathtt{metaterm}.\delta(\mathtt{def\_eqn}(l, r))$